Contents

INTRODUCTION

Getting Started with TradeStation EasyLanguage

The purpose of this book is to help you learn the basics of EasyLanguage and explain how to make use of this powerful language for creating your own trading strategies and analysis techniques using familiar terminology and simple logical statements. Please note this book is not intended to be a reference manual but is designed to be an introduction to EasyLanguage and its many uses.

Before proceeding, it's important that you have a working knowledge of the trading process and of TradeStation 6. The following assumptions have been made regarding the focus of this book:

- You are familiar with TradeStation and its features.
- You have trading experience and understand the basic technical analysis concepts used by TradeStation's built-in analysis techniques and commonly used strategy components.
- You are aware of the existence of additional sources of information (see "Additional Resources" on page 83).

We recommend that you read the chapters and perform the exercises in the order presented. As you proceed through the book, you should try out the examples using the sample data supplied with the product. This will ensure that you're properly introduced to the topics as they are needed and that your EasyLanguage learning experience will be both enjoyable and rewarding.

CHAPTER 1

The Language of Strategic Trading

This chapter introduces you to EasyLanguage and to some of the fundamental concepts associated with the analysis of charted data. This includes a discussion of how EasyLanguage processes instructions, and describes basic grammar and punctuation rules that apply to EasyLanguage.

The information in this chapter assumes that you have an understanding of the basic purpose and functionality of TradeStation. Since the topics in this chapter establish a foundation for the use of any EasyLanguage application in TradeStation, you may find that it's worth reviewing them even if you don't plan to immediately write your own strategies, functions, or analysis techniques.

In This Chapter

What is a Trading Strategy?

In TradeStation, a trading strategy consists of a set of objective rules that are used to describe when to buy and sell stocks, bonds, commodities, or other trading instruments. While most traders have some rules that guide their trading activities, these ‘rules’ are frequently based on subjective elements, such as intuition and emotion. The purpose of strategy trading is to be able to create a set of rules based on measurable factors, to verify that the rules work when applied to historical data, and to automate the rules so that your buying and selling decisions are based on an objective trading methodology.

It’s important to point out that TradeStation itself is *not* a trading strategy, but, rather, it is a sophisticated software tool that helps you implement trading strategies of your own design. TradeStation’s many powerful charting, analysis, and data collection features are all designed to help you refine your trading rules and develop your own strategies.

You’ll be using EasyLanguage to define the rules that reflect your trading ideas, and you’ll be using TradeStation to test and automate the strategies you create. As you automate your trading strategy with TradeStation, entry and exit orders are displayed on a chart based on the ideas you’ve developed (Figure 1-1).

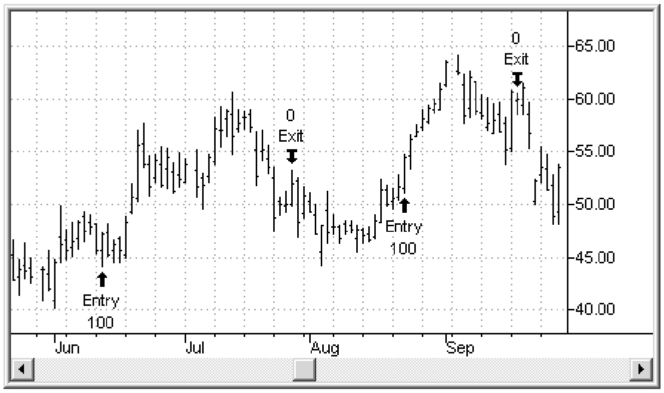


Figure 1-1. A trading strategy producing entry and exit orders.

One of the primary benefits that comes from developing your own trading strategy with EasyLanguage is the simple fact that you must write your rules down in a clear and concise manner. This helps eliminate ambiguous and inconsistent practices that every trader needs to guard against. In addition, creating a set of objective based rules can greatly reduce the negative effects that often come from emotional decision making.

Another advantage to strategic trading with TradeStation is that you are able to back-test your strategy on historical price data, which lets you know how your rules would have performed under changing conditions. After all, planning ahead and being ready to act are what strategic trading is all about. And, by automating your trading with a tested strategy, you’ll have more time to spend doing research and developing new trading ideas.

As you proceed through the following chapters, you’ll be exposed to a variety of approaches and examples that can help you develop your own trading ideas.

What is EasyLanguage?

Since TradeStation, and the computer it runs on, can't read your mind (not yet at least), the trading rules for your trading strategy need to be written in a form that both you and TradeStation can understand. EasyLanguage was developed by TradeStation Technologies especially for this purpose.

EasyLanguage is a simple, but powerful, computer language for producing objective rules and calculations that are used to create trading and technical analysis tools. By combining common trading terminology with simple decision statements, EasyLanguage makes it easy for you to write your own trading rules and actions in a clear and straightforward manner. TradeStation reads your EasyLanguage statements, evaluates them based on the price data that has been collected, and performs the specified actions.

EasyLanguage is designed so that traders can write their ideas in plain English, using trading terms and phrases with which they are already familiar. For example, compare the following two statements of the same trading idea regarding your favorite stock - the first as if you had jotted the idea down on paper and the second as it might appear in EasyLanguage.

As jotted down on a piece of paper:

> "if the close is greater than the high of 1 day ago, then buy 100 shares at market"

As written in EasyLanguage:

```
if the Close > the High of 1 day ago then

      Buy 100 shares next bar at market ;
```

Not much difference, right?

So, while this example may not represent the most sophisticated trading idea, you can see that EasyLanguage truly does allow you to make your instructions very readable. In the next chapter, you'll be learning much more about how to write EasyLanguage instructions that perform desired trading actions based on your ideas.

In addition to letting you develop trading strategies, EasyLanguage is also used to create your own custom analysis techniques or functions. Or, if you choose, you can copy and modify any of the hundreds of built-in analysis techniques, functions, and commonly used strategy components that are part of TradeStation.

The important thing to remember is that EasyLanguage is not only the language of TradeStation, it's the language of strategic trading!

Scanning the data on a chart

Before you can understand how to write a trading strategy or analysis technique in EasyLanguage, it's important to review exactly how EasyLanguage operates.

In TradeStation, a chart typically consists of numerous bars built from price data associated with a specified symbol. Each bar summarizes the prices for a trading interval, most commonly a time period such as five minutes or one day, and includes values such as the open, high, low, and closing prices for the period. Other bar data such as the traded volume and the date/time of the bar's close is also available.

Since one of the primary purposes of EasyLanguage is to look at price data from one bar and compare it to data from other bars, you need to understand how your EasyLanguage procedure (indicator, ShowMe, trading strategy, etc.) reads the price data from a chart.

In this simple one line trading strategy:

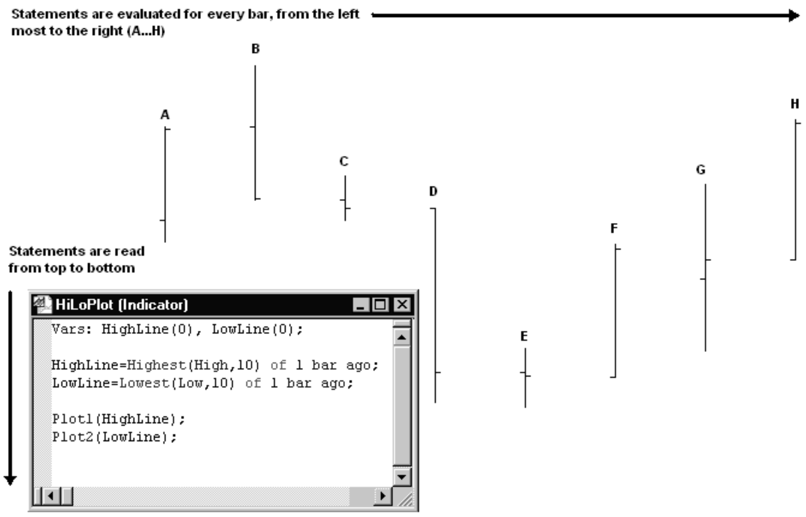
```
if the Close > High of 1 bar ago then Buy next bar at market;
```

you are instructing EasyLanguage to compare the closing price of one bar with the high price of another and to generate a buy order when the close is greater than the high. This comparison is made on the closing price of every bar in the chart, each time looking at the high price from the bar before.

Even though your EasyLanguage analysis technique is applied to a chart filled with bars, the process used to evaluate the data on the chart is always the same. Remember, a chart is simply a visual representation of a period of trading history for a symbol, where individual bars represent trading intervals. Each bar contains basic price data (prices, volume, date, etc.) that was saved from a datafeed. To evaluate your chart, EasyLanguage turns back the clock and starts reading the price data from the first bar in the chart just as it appeared from the datafeed when that bar was created. In terms of your EasyLanguage procedure, this is now the *current bar*. The EasyLanguage statements in your procedure are always evaluated relative to the current bar. On the first bar, there are no previous bars and the comparison in the example above cannot be true. When your procedure is done evaluating the bar, EasyLanguage steps forward in time to the next bar, making it the *current bar* on which the statements in your procedure are evaluated.

Typically, an EasyLanguage procedure includes a number of statements, each of which can result in an action such as plotting a line on the chart or generating a buy/sell short order. After all of the statements in the EasyLanguage procedure are processed for the current bar, the price data from the next bar is read and the procedure is run again using the new prices. This continues, across the chart from left to right, until all of the prices from all of the bars on the chart have been read. The result is that, for a 500 bar chart, the EasyLanguage procedure runs a total of 500 times, once on each bar.

For example, look at the chart in Figure 1-2, consisting of bars A through H, where the indicator "_HiLoPlot" has been applied. Each line of the indicator, numbered 1 through 5, is evaluated on every bar, starting with the price data from bar A, then from bar B, etc., across all of the bars in the chart. Even though you may not understand the EasyLanguage statements at this time, it's important to know that each statement is evaluated, in order, for every bar.


Statements are evaluated for every bar, from the left most to the right (A...H)
A
B
C
D
E
F
G
H
Statements are read from top to bottom
HiLoPlot (Indicator)

```
Vars: HighLine(0), LowLine(0);

HighLine=Highest(High,10) of 1 bar ago;
LowLine=Lowest(Low,10) of 1 bar ago;

Plot1(HighLine);
Plot2(LowLine);
```

Figure 1-2. Evaluating bars from left to right.

Reserved Words

In EasyLanguage, just like any other language, words have meanings and are combined into statements using some type of grammatical structure. Punctuation marks are used to signify the end of each statement and to separate phrases within each statement. The basic vocabulary of EasyLanguage consists of a set of reserved words, each having a specific purpose, such as to compare and evaluate expressions, to specify display or trading actions, and to reference values. In the following chapters you'll learn more about these words and how to use them to build your analysis techniques.

Price Data

The ability to evaluate price data is one of the most important elements of EasyLanguage. As a result, a number of reserved words exist in EasyLanguage that refer to the price data available from each bar. The words typically match the common trading term for the same value, such as Open, High, Low, Close, or Volume. Table 1-1 lists some of the most frequently used price data values:

Data Word	*Abbreviation*	*Description*
Open	O	First available price for the bar
High	H	Highest price within the bar
Low	L	Lowest price within the bar
Close	C	Last available price for the bar
Date	D	Date of the last trade within a bar
Time	T	Time of the last trade within a bar (in 24 hour format)
Volume	V	Total volume of trades within the bar
OpenInt	I	(Open Interest) Total number of open contracts

Table 1-1. Frequently used reserved words for price data.

For example, the reserved data word Close refers to the closing price of the bar currently being evaluated by the EasyLanguage procedure. Remember that your EasyLanguage procedure is applied to each bar on the chart, from left to right, and that the ‘current bar’ is always the bar on which your procedure is running. If your procedure is running on the 7th bar of the daily chart, the High reserved word contains the high price for the 7th day of trading on the symbol being charted.

Since trading decisions are rarely made on just one bar’s worth of price information, EasyLanguage makes it easy to get price data from any bar older than the current bar by simply adding the phrase ‘of N bars ago’ after the appropriate reserved word.

For example, the EasyLanguage expression ‘Low of 1 bar ago’ refers to the low price of the previous bar (relative to the bar currently being evaluated by EasyLanguage). In a similar example, if your EasyLanguage procedure is running on the 12th bar of your chart, the expression ‘Volume of 3 bars ago’ refers to the charted symbol’s volume from the 9th bar, or 3 bars back from the current bar. The alternate method for referring to data from a previous bar is to use square brackets ‘[N]’ after the reserved word – such as, Open[2] to refer to the opening price from 2 bars ago.

In order to remain efficient when analyzing charts containing hundreds or thousands of bars, EasyLanguage contains a special setting called *MaxBarsBack* that is used to identify how many previous bars of price data an EasyLanguage procedure can reference.

For example, if you write an EasyLanguage procedure that uses a 14-bar moving average, your procedure needs to have at least 14 bars of data to perform its calculations. By setting *MaxBarsBack* to 14, in this case, your procedure would wait until 14 bars have passed (from left to right) to be sure that enough data is available to calculate the 14-bar moving average for the current bar. EasyLanguage would do the same for each current bar throughout the rest of the chart.

The rule is that *MaxBarsBack* must be equal to or greater than the largest value needed to perform the analysis. For example, if you are calculating an index based on 60 days of price data, then you’ll require that *MaxBarsBack* be set to 60 or greater.

To make it easy on both the developer and end-user, most EasyLanguage analysis techniques automatically calculate the *MaxBarsBack* value. This is done by selecting the **Auto-detect** option under the heading 'Maximum number of bars study will reference' on the **General** tab of the **Format [Analysis Technique]** dialog box. In the Auto-detect mode, EasyLanguage evaluates all of the data references in your procedure and automatically sets the optimal value for *MaxBarsBack*. For more information, search the TradeStation Help for the phrase *Maximum number of bars*.

Statements

Those EasyLanguage reserved words that perform comparisons, carry out associated actions, and control other program operations are called statements.

These include the *If-Then* structure, the *Plot* statement, and variable declaration statements. Just like a sentence represents a complete thought in the English language, an EasyLanguage statement represents a complete instruction that results in some program action. You'll be introduced to all of the basic EasyLanguage statements later in this book.

Skip words

To make EasyLanguage read more like English, another group of reserved words called *skip words* are provided. These optional words, such as *the, at, on,* and *from,* can be included in a statement or expression.

For example, the following:

```
if Close > High[1] then Buy next bar at market;
```

could also be written using skip words to make it appear more readable:

```
if the Close > the High of 1 bar ago then
   Buy on the next bar at the market;
```

Be aware that, while making your EasyLanguage instructions easy to read, skip words perform no action within the actual program. In other words, they are ignored when the procedure is run. Whether you use skip words at all is a matter of personal preference.

The following is a list of skip words:

a	by	of	the
an	does	on	was
at	from	place	
based	is	than	

Table 1-2. Skip Words.

Punctuation

While sentences in the English language are separated from one another using a period (.), EasyLanguage uses the semicolon (;) to mark the end of each statement. Statements can be very simple, such as:

```
Plot1( High, ”HighPlot” ) ;
```

or more complex multi-line expressions like this:

```
if the Close > High of 1 bar ago + ( High - Low ) / 2  and
   Average( Volume, 3 )[1] < Volume then Buy next bar at market;
```

Even though the second example includes several calculations and conditional expressions, both examples are valid statements that start with a statement reserved word and end with a semicolon (;). In addition to the end-of-statement punctuation mark, there are several other punctuation symbols shown in these examples. Because these symbols hold a special meaning in EasyLanguage, they are also considered reserved words. You’ll be using these often when writing in EasyLanguage.

Table 1-3 lists the punctuation marks used in EasyLanguage.

Symbol	Name	Description
;	Semicolon	Ends a statement
( )	Parentheses	Groups values that should be calculated together
,	Comma	Separates items in a list, such as parameters used with functions
“ “	Quotation marks	Used to indicate text items
[ ]	Square brackets	References price data from previous bars and array elements
{ }	Curly brackets	Used to write comments about the operation of your EasyLanguage statements

Table 1-3. EasyLanguage punctuation marks.

Summary

While there are many more reserved words and symbols in EasyLanguage, the important thing to remember is that they all perform a specific role and must be used according to the rules that are defined for the language.

During the next several lessons you’ll learn all about these rules and how to write your own EasyLanguage instructions based on these reserved words and statements.

Exercises and Review

Review

Trading Strategies consist of a set of rules and actions, written in EasyLanguage, that produce entry and exit orders based on your own trading ideas.

EasyLanguage is the language of strategic trading. Using common trading terminology, it lets you evaluate market conditions and produce trading actions.

Reserved words in EasyLanguage include all statements, skip words, and punctuation marks.

Bars on a chart are evaluated from left-to-right, and EasyLanguage procedures look at every bar.

Exercises

(Answers are contained in Appendix A)

I. Match each numbered word with its correct definition. Write the matching letter next to the word's number.

1. Reserved Words	A. Ignored during execution
2. Statement	B. Indicator, Study, or Strategy
3. MaxBarsBack	C. The Language of Strategic Trading
4. Price Data	D. Runs on each bar
5. Skip Word	E. Basic vocabulary of EasyLanguage
6. Semicolon	F. A complete EasyLanguage instruction
7. Procedure	G. Number of bars ago that can be referenced
8. Analysis Technique	H. Ends a statement
9. EasyLanguage	I. Values associated with each bar

II. Mark the following either True or False (T or F).

1. EasyLanguage only evaluates a bar when the price changes.
2. All EasyLanguage reserved words are statements.
3. Bars are evaluated from left to right.
4. Skip words automatically jump to the next statement.

CHAPTER 2

Your First Trading Strategy

In this chapter you'll learn how to convert simple trading rules into EasyLanguage statements in order to create trading strategies. You will be introduced to the construction and use of conditional expressions to make comparisons that notify you when to place a trade. In the process, you'll learn about variables, functions, and other EasyLanguage components that make your strategy more flexible and easier to understand.

Since the material in this chapter builds on a general understanding of the vocabulary and punctuation of EasyLanguage, it is recommended that you read the previous chapter and complete the exercises at its end. We also assume that you are familiar with the Chart Analysis window and price data.

In This Chapter

Using the PowerEditor

The PowerEditor is a full-featured editor for creating and modifying EasyLanguage instructions. In addition to providing common word-processing features for editing your EasyLanguage procedures, it includes specialized features that color-code the various elements of your statements and automatically check your work for proper syntax.

While it is fairly simple to use, if you are not familiar with the behaviors of PowerEditor, you should read about it in the TradeStation Help before continuing with the following material.

Comparisons and Conditions

One of the essential ingredients of a trading strategy is the ability to respond to price changes in the market and to perform a trading action (e.g., buy, sellshort, and/or exit) based on your trading ideas. An EasyLanguage trading strategy looks at the data from each bar in a chart and, typically, compares the current bar's price data with that from previous bars.

In this section you'll learn how to translate trading ideas into rules and comparisons so that EasyLanguage can perform the desired trading actions.

Simple Expressions

The first step in translating your ideas into EasyLanguage is to create one or more 'rules' that test for pre-determined market conditions. When the conditions that make up the rule are judged to be true, EasyLanguage performs the trading action you specify.

If...Then

The most commonly used EasyLanguage instruction for making comparisons is the *If...Then* statement. The condition to compare is stated following the word *If* and the action to be taken follows the word *Then*. A condition can be a simple comparison of two values or can be a complex combination of multiple calculations and conditions.

The following simple example tests to see if the current bar's closing price is greater than the high price of the previous bar:

```
if Close of this bar > High of 1 bar ago then
   Buy next bar at market ;
```

... and results in a buy order when the condition is true.

In the example above, the section between the words *If* and *Then* is called a *conditional expression* that consists of the values separated by a *relational operator*. Don't let the names scare you! A relational operator is nothing more than a symbol or phrase that specifies how to compare the first value with the second in a conditional expression. EasyLanguage tests the values against one another, and if they match the stated comparison, then the condition is said to be *true*.

For example, the condition (highlighted in grey):

```
if High = High of 1 bar ago then ACTION ;
```

... is true if the high price of the current bar equals the high of one bar ago.

Likewise, the conditional expression (highlighted in grey):

```
if the Close is > the Open then ACTION ;
```

... is true if the current bar's closing price is greater than its opening price.

The second part of the *If…Then* structure consists of an ACTION which represents any valid EasyLanguage statement. The condition must be true for the action statement following the word *Then* to be evaluated by EasyLanguage.

Table 2-1 contains a list of the basic EasyLanguage *relational operators* and the condition each represents:

Operator	*Meaning*
=	Equal to
<>	Not equal to
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to

Table 2-1. Relational Operators.

In addition to the basic relational operators, EasyLanguage also provides another pair of operators that are useful for comparing prices, or moving averages for example, that cross one another. The *Crosses Over* and *Crosses Under* relational operators (Table 2-2) compare prices on the current bar and those of the previous bar to see how they have changed. You'll be using this type of comparison in a later example.

Operator	*Meaning*
Crosses Over or Crosses Above	X *Crosses Over* Y True when X is greater than Y on the current bar after being less than or equal to Y on the previous bar
Crosses Under or Crosses Below	X *Crosses Under* Y True when X is less than Y on the current bar after being greater than or equal to Y on the previous bar

Table 2-2. Additional relational operators.

Buy/SellShort

In a trading strategy, some of the most common actions are the statements *Buy* and *SellShort* which are used to enter a trading position for your selected symbol.

For example, in the following:

```
if the Close is > the Open then Buy next bar at market ;
```

a buy order would be generated if the current bar closes above the open (making the condition true).

Similarly, in the following:

```
if the Open is > Close then SellShort next bar at market ;
```

a sellshort order would be generated if the bar's open is above its close.

You'll learn more about entry orders later in this chapter. For now, all you need to worry about is that you can place a buy or sellshort order as a result of a condition being true.

Now it's your turn. Periodically, you'll be asked to create an example of an EasyLanguage procedure in the PowerEditor and apply it to a chart. This will help you become familiar with the process of editing EasyLanguage instructions and running them.

The first example you're going to create is a *strategy*. Typically, a strategy is based on a set of rules that determines when and how to enter or exit a trade. In this example, the strategy is based on the idea that you want to buy whenever a symbol closes above the previous bar's high, indicating upward price activity. Since the price data you'll be using in this chapter is for a daily stock, this strategy will generate a buy order when the stock closes higher than yesterday's high.

In TradeStation, use the **File - New** menu sequence, click on the **EasyLanguage** tab, and select **Strategy** to create a PowerEditor Strategy Document. Give it the name *_CloseUp*, and for our purposes select (None) for the template. (**Note**: You'll use the underscore character in front of the name for most examples in this book so that they'll appear at the top of the list when selecting them). You should now have a blank window titled *TradeStation EasyLanguage PowerEditor - **Strategy:** _CloseUp*.

Type the following statement into the PowerEditor window:

```
if Close of this bar > High of 1 bar ago then
   Buy next bar at market ;
```

Example 2-1. Strategy named *_CloseUp*.

Notice that the color of words change as you type. For example, reserved words appear in one color, skip words in second color, and other words in a third color. Using this feature can help you identify misspelled or incorrect words in addition to making your EasyLanguage instructions more readable.

Before continuing, make sure that you remembered to type the semicolon at the end of the statement.

Now, click the **Verify** button from the PowerEditor toolbar.

After a few seconds, a *Verification successful* message should appear in the middle of your screen. Your EasyLanguage strategy is now ready to go. If you made any entry mistakes, an error message would appear (error messages can be viewed in the **Verify** tab of the EasyLanguage Output Bar.

In TradeStation, create a Chart Analysis window to this new workspace using symbol MSFT (Microsoft). Set the chart interval to **Daily** and the **Days Back** value to 500. For more information on this procedure, search the Help Index.

From the chart, use the **Insert - Strategy** menu sequence. Add the strategies named _*CloseUp* and *TimeExit (Bars) LX* to the chart using all the default settings.

***Note**: The **TimeExit (Bars) LX** strategy closes out a long position after 5 days so that you can see many occurrences of the example order.*

Now, you should see up arrows with the word **Buy** underneath (Figure 2-1) pointing to bars after a **Close** price is greater than the previous bar's **High**.

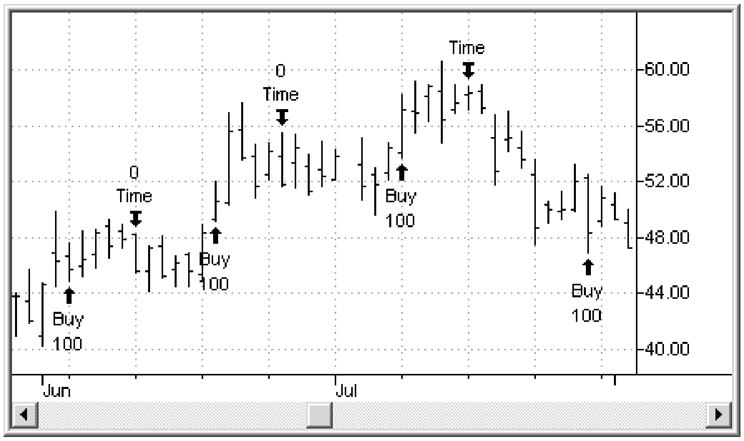


Figure 2-1. Strategy _*CloseUp* and *TimeExit (Bars) LX* .

If you look closely, you'll notice that the buy arrow and price marker (pointing to the bar's opening price) appear on the bar *after* the condition is met. That's because your order said to buy on the next bar at the market price (assumed to be the Open). Also, notice the *Time* label five bars after each buy. That's the exit strategy *TimeExit (Bars) LX* closing out your long position after five days. You may also notice quite a few bars that do not have buy arrows, even though the **Close** was greater than the previous **High**. That's because the default setting for a strategy is to allow only one position at a time in a particular direction. You can modify this action by changing the *Position Limits* settings on the **General** tab of the **Format Strategy** dialog box. For more information, search the TradeStation Help.

The following example is a variation of the previous strategy:

```
if Close > High[1] then Buy next bar at market ;
```

Example 2-2. Strategy named *_CloseUp2*.

It performs exactly the same comparison, but uses a more concise syntax. Note the use of the square brackets to specify *1 bar ago* and that the skip words have been removed. Try creating a new strategy named *_CloseUp2,* type in the new statement above, and verify the strategy. Then, create another Chart Analysis window and apply the strategy to your chart. You should see exactly the same Buy and Exit markers on your chart, since strategy *_CloseUp2* performs the same comparison as *_CloseUp*.

Calculations

In the previous example, the values being compared are individual bar prices (*Close* and *High*) which are EasyLanguage reserved words. However, values can also be the result of calculations on either side of the relational operator.

In EasyLanguage, *mathematical operators* perform addition, subtraction, multiplication, and division on a set of values. The symbols for these operators are:

Operator	*Meaning*
+	Addition
–	Subtraction
*	Multiplication
/	Division

Table 2-3. Mathematical operators.

For example, to determine the range of a bar (which is the difference between the bar's highest and lowest price) you would subtract the Low price from the High. Therefore, the expression for the current bar's range would look like this:

```
High - Low
```

which would be stated in English as "the High minus the Low." And the expression for the range of a previous bar would be:

```
High[1] - Low[1]
```

and would read "the High of 1 bar ago minus the Low of 1 bar ago."

How would you calculate a value equal to 50% of the previous bar's range? In English you might say "take the previous bar's High minus the Low and divide by 2." You might be tempted to write it in EasyLanguage as: High[1]-Low[1]/2. However, you'd be wrong! Remember that the value you want is 50% of the difference between the prices and not the High minus 50% of the Low. In EasyLanguage, and most other computer applications, multiplication and division are performed before addition and subtraction.

Therefore, you need to control the order of calculations through the use of parentheses because a parenthetical calculation is performed before all others. In the last example, the correct EasyLanguage expression would be:

```
( High[1] - Low[1] ) / 2
```

where the subtraction within the parentheses is done before the division, resulting in the proper value.

Now, let's get back to creating another strategy where a buy orders are generated if the Close is greater than the previous bar's High plus 50% of the previous bar's range (the calculation you just developed).

Your new strategy will look like this:

```
if Close > High[1] + (High[1]- Low[1])/2 then
   Buy next bar at market ;
```

Example 2-3. Strategy named *_CloseUp3*.

In the PowerEditor, open the Strategy file named *_CloseUp2* and insert the highlighted portion from above. Save the revised strategy as *_CloseUp3*. Then, in TradeStation, apply the strategies *_CloseUp3* and *TimeExit (Bars) LX* to your sample chart (Figure 2-2).

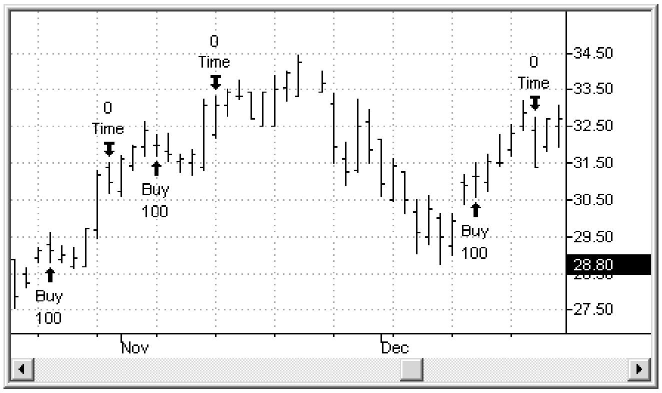


Figure 2-2. Strategies *_CloseUp3* and *TimeExit (Bars) LX*.

Compound Expressions

In the previous section, you were introduced to simple expressions, which consist of a single comparison. Regardless of the complexity of the calculations expressed on either side of a relational operator, a simple expression only compares two values. In this section, you'll learn about compound expressions that contain multiple comparisons.

Condition Variables

In the last example, you may have noticed that the *If…Then* statement was starting to get hard to read because the calculation was getting longer. EasyLanguage offers a solution …you can give the condition a name and save its true/false value for later use. For your convenience, EasyLanguage reserves the names *Condition1* through *Condition99* for this purpose. All you need to do is assign an expression to the *ConditionN* variable and use the variable in the *If…Then* statement. For example, the following two lines:

```
Condition1 = Close > High[1] + ( High[1] - Low[1] ) / 2 ;
if Condition1 then Buy next bar at market ;
```

are exactly the same as the single statement below:

```
if Close > High[1] + ( High[1] - Low[1] ) / 2 then
   Buy next bar at market;
```

Notice that the *If…Then* statement in the first example is much easier to read. You'll learn more about variables and assignment statements later in this chapter.

AND - OR

Often, it is desirable to combine multiple comparisons in a statement. For example, building on the previously created strategy, you may want to buy based on an increase in the Volume of trades in addition to a higher price.

In EasyLanguage, you can combine several conditions in an *If…Then* statement. The reserved words, AND and OR are used to create a compound expression. For example, in the following:

```
if Condition1 and Condition2 then Buy next bar at market;
```

the compound expression is true if both the first condition AND the second condition are true. If either *Condition1* or C*ondition2* is false, the entire compound expression is false.

Just the opposite is true for the OR reserved word. In the following example:

```
if Condition1 or Condition2 then Buy next bar at market;
```

the compound expression is true if either the first condition OR the second condition is true. If both *Condition1* and *Condition2* are false, the entire compound expression is false.

When creating a compound expression with both the AND and OR reserved words, you need to use parentheses to organize the terms of the expression so that the conditions are evaluated in the proper order. For example, the following statement:

```
if Condition1 and Condition2 or Condition3 then
   Buy next bar at market ;
```

is unclear because it has two possible combinations that make it true. See if you can figure them out.

To make the previous example clear, you need to use parentheses. Here are the two possible valid combinations:

```
if ( Condition1 AND Condition2 ) or Condition3 then
   Buy next bar at market ;
```

and

```
if Condition1 and ( Condition2 or Condition3 ) then
   Buy next bar at market ;
```

Take a moment to make sure that you understand how these differ before you try to use multiple conditions in a compound expression.

Now, back to work.

For the next example, you'll be creating a strategy that buys into the market based on the combination of two ideas. The first idea is based on the previously developed strategy where you tested to see if the current bar's close was greater than the previous bar's high by at least 50% of the bar's range. The second idea is to test whether the trade volume on the current bar is greater than the previous bar's volume by at least 50%. You want to generate a buy order when both conditions occur.

Create a new strategy named *_CloseUpAndVolumeUp* and enter the following EasyLanguage instructions:

```
Condition1 = Close > High[1] + ( High[1] - Low[1] ) / 2 ;
Condition2 = Volume > Volume[1] * 1.5 ;
if Condition1 and Condition2 then Buy next bar at market ;
```

Example 2-4. Strategy named *_CloseUpAndVolumeUp*.

The first condition is the same as the one you developed in the previous *_Closeup3* strategy and tests for a price rise. The second condition is used to test for an increase in the level of trading activity. When both conditions are true, a buy order is generated. In this simple example, you clearly see how one idea can be used to confirm another, which demonstrates an important principle in strategy development.

Now, apply this strategy to your sample chart (along with *TimeExit (Bars) LX*) and observe the buy orders.

More About Variables

Variables are used to save values that you will use later in your procedure to help make your EasyLanguage instructions easier to read and understand. To do this in EasyLanguage, you use an *assignment statement*, which begins with a variable name followed by an equal sign and a value or expression to be saved.

True/False and Numeric

When you perform a comparison in EasyLanguage, the result is a true/false value. In the previous section, you learned about assigning a true/false value to a *ConditionN* variable for use in an *If…Then* statement.

You can also assign, or save, the numeric result of a calculation to a variable in exactly the same way. The reserved words *Value1* through *Value99* are available for this purpose. For example, in the following:

```
Value1 = ( High[1] - Low[1] ) / 2 ;
```

the numeric result of the calculation to the right of the equal sign is saved as *Value1*. If you use *Value1* in another expression, such as:

```
Close > High[1] - Value1 ;
```

the number saved as *Value1* in the previous assignment is used in the calculation.

Remember, a *ConditionN* variable is used to save a true/false value, and a *ValueN* variable is used to save a numeric value. These reserved variables are automatically initialized to *false* for *Condition1* through *Condition99* and to 0 for *Value1* through *Value99*.

Declaring Your Own Variables

In addition to using the standard EasyLanguage variable names (*ConditionN* and *ValueN*) to save true/false and numeric values, you can create your own variable names. Instead of trying to remember the difference between *Value2* and *Value43* in a calculation, you could use more meaningful names such as *FiveBarHigh* or *UpDayCount* for your saved values.

Before you can assign a value to your own variable, EasyLanguage must know its name. This is done using a *variable declaration* statement. The reserved word "Variable" is used to declare the name of a variable. As part of the declaration, you must include the initial value for the variable within the parentheses following its name. Based on the initial value you declare, the type of the variable will be either numeric or true/false. For example, the following:

```
variable: BarRange(0) ;
variable: PriceUp(False) ;
```

declares two variables, one named *BarRange* and the other named *PriceUp*. The first variable is numeric, since its initial value is the number "0", and the second variable is true/false, since its initial value is the condition "False." In addition to being used to specify the type of variable, the initial value also sets the starting value of the variable for the first bar.

You can also use the reserved words "Var" or "Variables" instead of "Variable" to begin the declaration statement. Also, instead of using separate declarative statements for each variable, you can use a single declaration statement to declare a number of variables at the same time by separating the names with commas, such as:

```
variables: BarRange(0), PriceUp(False), BuyPrice(50) ;
```

Remember, each declaration statement must end with a semicolon.

In the last strategy that you created, a part of the calculation included the value of a bar's range (the High minus the Low). The following strategy uses a numeric variable named *PrevBarRange* to save the previous bar's range and use it in a calculation:

```
variable: PrevBarRange(0) ;

PrevBarRange = High[1] - Low[1] ;
if Close > High[1] + PrevBarRange / 2 then
   Buy next bar at market;
```

Now, here's another strategy that does exactly the same thing as above, but with one important change that shows the real power of EasyLanguage:

```
variable: BarRange(0) ;

BarRange = High - Low ;
if Close > High[1] + BarRange[1] / 2 then
   Buy next bar at market ;
```

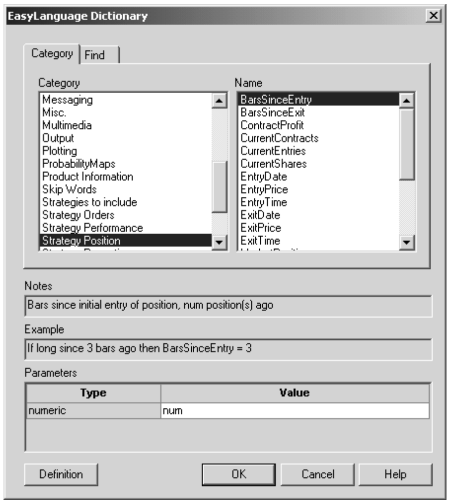
Notice that the variable *BarRange* is assigned the value of the current bar's range. However, the *If…Then* comparison refers to the value of the previous bar's range by nature of the "[1]" following the variable's name. That's right, even variables that *you* create can reference values from previous bars!

EasyLanguage Dictionary

Instead of having to remember hundreds of reserved words, you can use the built-in EasyLanguage Dictionary, accessable from the PowerEditor, to look up and paste them into your procedure. The EasyLanguage Dictionary also includes information about any parameters and data types (numeric, true/false, etc.) that are associated with the word.

Categories and data types

Reserved words in the EasyLanguage Dictionary are organized by category to make it easy to locate a particular word. Simply click on a category name (in the left portion of the EasyLanguage Dictionary window) to see the associated words under that category. Click on a word (in the right portion of the window) to see a brief description of its meaning. For a detailed description, click the **Definition** button in the lower left corner of the **EasyLanguage Dictionary** dialog box (Figure 2-3).



Bars since initial entry of position, num position(s) ago

If long since 3 bars ago then BarsSinceEntry = 3

Type	Value
numeric	num

Figure 2-3. EasyLanguage Dictionary.

For example, select the category “Strategy Position” and click on the word “BarsSinceEntry” to see its meaning. To insert the word into your EasyLanguage procedure, click the **OK** button.

Try looking up several words in the same category, such as “MarketPosition” and “EntryPrice." Then paste them into your procedure. Notice that the word and its parameters appear in the PowerEditor.

In upcoming examples, you’ll be using EasyLanguage words, such as *MarketPosition*, that provide status about your strategy or order. These and many more values can be found in the EasyLanguage Dictionary and in the TradeStation Help.

What is a Function?

Up to this point, you’ve created strategies that use comparisons and calculations based on individual prices such as **Close** or the **High of 2 bars ago**. In the world of trading, however, it’s very common to base your ideas on a range of prices, such as the average **High** of the last 10 bars, or on the value of a common analysis calculation, such as the Relative Strength Index (**RSI**). To support this, EasyLanguage lets you refer to secondary calculations, called *functions,* that can be used in comparisons and calculations much like variables. Each function has a name and returns a value based on some underlying calculation.

EasyLanguage includes a large number of built-in functions, including common trading indexes and price calculations. Much like with reserved words, functions can be accessed from the EasyLanguage Dictionary. For example, one such function returns the highest value of a particular price across a range of bars. You specify the price you want to test and the number of bars back you want to test as follows:

```
Value1 = Highest(Close, 5) ;
```

The values enclosed in parentheses are called parameters. The *Highest* function has two parameters; the first specifies what price to look at and the second indicates how many bars back to test. In this example, *Highest* looks for the highest **Close** price over the last **5** bars so that it can be assigned to variable *Value1*.

Although it is used much like a variable, a function has three important differences:

1. A function does not have to be declared.
2. You cannot assign a value to a function. A function returns a value based on calculations that are defined when the function is created.
3. The same function can be referenced from many different trading strategies and analysis techniques. You'll learn more about this in future examples.

In addition to the large library of standard functions, EasyLanguage also lets you write your own functions based on calculations and parameters that you define. This powerful feature allows you to create your own custom library of functions that might include the most popular new market index or a set of time tested calculations that you've been trading with for years. You'll learn more about writing your own functions in a later chapter.

Using a Function

Now, let's look at writing a strategy that uses a function.

This strategy will look for the start of an uptrend, such as when a fast moving average (short time-frame) crosses over a slow moving average (long time-frame), and will place an order to buy. You'll use the *Average* function from the EasyLanguage Dictionary along with the *Cross Over* relational operator as the basis for your comparison.

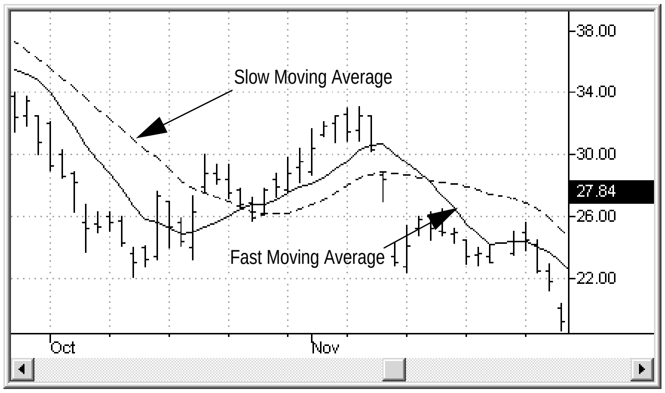


Figure 2-4. Chart with indicator showing Fast and Slow Moving Averages.

Create a new strategy named *_MovAvgUp* and type the following statement:

```
if Average(Close, 9) crosses over Average(Close, 18) then
    Buy next bar at market ;
```

The *Average* function requires two parameters; the first is the price (open, high, close, etc.) and the second is the number of days back on which to calculate the average. In this example, *Average* is used to calculate both the fast moving average (9 days) and the slow moving average (18 days) of the closing prices on your chart. When the fast moving average value crosses over the slow, a buy order will be placed at the market price for the next bar.

After you verify your strategy, apply *_MovAvgUp* and *TimeExit (Bars) LX* to your sample chart (Figure 2-5) and you should see a buy order arrow near the start of each up trend.

In fact, because it is based on the fast moving average, the buy order will appear approximately 9 bars after the trend starts.

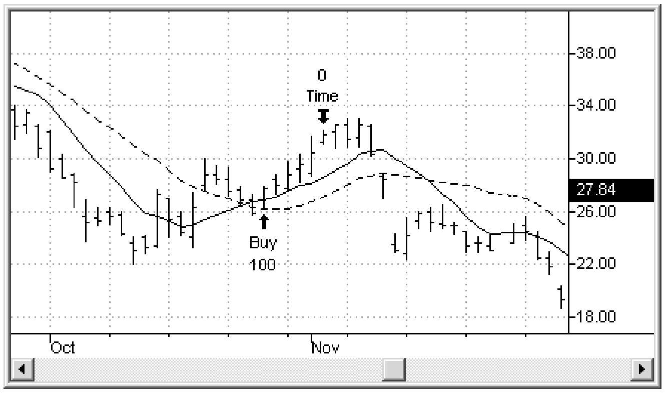


Figure 2-5. Strategies *_MovAvgUp* and *TimeExit (Bars) LX* on a chart with the *Mov Avg 2 Lines* indicator.

Inputs

Instead of using fixed values for the fast and slow moving average lengths in the *_MovAvgUp* strategy, wouldn't it be nice if you could change these values at the time you apply a trading strategy to a chart. Well, you're in luck, because EasyLanguage lets you do just that!

The *Input* statement allows you to declare a named value that can be changed when you apply the strategy or analysis technique to a chart. You can use *Inputs* in the strategy you just created to allow the number of days for the fast and slow moving average to be changed by the user instead of using the fixed values of 9 and 18.

Using Inputs

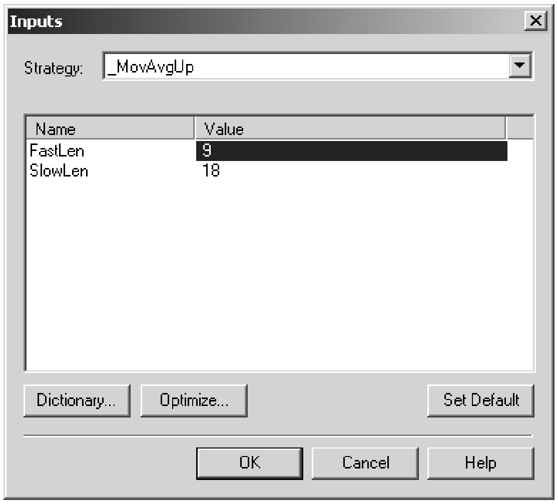
Go back and open the previously created strategy *_MovAvgUp*. Make the following EasyLanguage changes so that the strategy looks like this:

```
inputs: FastLen(9), SlowLen(18) ;

if Average(Close, FastLen) Crosses Over Average(Close, SlowLen)
   then Buy next bar at market ;
```

Much like when declaring a variable, you declare the name of each input along with its initial value (in parentheses). Now you can use the named values in your calculations just as you would the number. In this case, *FastLen* replaces the number 9 in the first average and *SlowLen* replaces the number 18 in the second average.

Using the **File - SaveAs** menu sequence, create a new strategy named *_MovAvgLength*. Go back to your chart and apply the strategies *_MovAvgLength* and *TimeExit (Bars) LX* to your chart. This time, notice that under the **Inputs** column on the **Format Strategy** dialog box there are values for the strategy *_MovAvgLength*. By selecting the strategy and clicking the **Inputs** button, you can change the values of *FastLen* and *SlowLen* (Figure 2-6). For more information on formatting inputs, see the TradeStation Help.



Strategy: _MovAvgUp

Name	Value
FastLen	9
SlowLen	18

Figure 2-6. Format Strategy Inputs dialog box

In addition to increasing the flexibility of your strategy, inputs allow you to use the optimization feature to determine the optimal values for each input. For more information, search the TradeStation Help.

Multiple Conditions and Actions

Earlier, you learned how the *If…Then* statement is used to perform an action whenever a condition is true, such as:

```
if Close > High[1] then Buy next bar at market ;
```

But what if you want to have EasyLanguage perform more than one action when a condition is true, such as generating a buy order and changing the value of a variable?

Begin...End

The reserved words **Begin** and **End** let you perform multiple actions with an *If…Then* statement. In EasyLanguage, a group of statements bounded by the words *Begin…End* is called a *block statement*. For example, the following strategy:

```
variable: CountTheBuys(0) ;

Condition1 = Close > High[1] + Range[1] ;
if Condition1 then
   begin
   Buy next bar at market ;
   CountTheBuys = CountTheBuys + 1 ;
   end ;
```

...evaluates the condition and, if true, places a buy order and increments (adds one to) a variable that counts the number of buys placed by this strategy. You can include any number of statements between *Begin* and *End*. The statements are only executed if the condition is true. If the condition is false, EasyLanguage skips to the word *End* and then continues. Also, note that it's common practice to indent the individual statements in a block, but it is not required. The indents simply make it easier to read and understand that the block is processed based on the *If* condition.

Frequently, you'll want to have one condition evaluated only after another is true. For example, you might want to check if you have a position in the market before evaluating an exit condition. The concept of combining one or more *If...Then* within another *If...Then* is called *nesting*. In the following:

```
variable: BarCounter(0) ;

if MarketPosition <> 0 then
   begin
   BarCounter = BarCounter + 1 ;
   if Close < Close[1] then
      begin
      Sell next bar at market ;
      BuyToCover next bar at market ;
      end ;
   end;
```

the first *If* condition is true if you have a position in the market from a previous buy or short sell. If the first is true, the following occurs: 1) the variable *BarCounter* is incremented, and, (2) the second *If* compares the current close with that of the previous bar and exits long and short if the condition is true. If either the first or second conditions are false, no action is taken. You can nest as many *If...Then* conditions as you choose.

***Note**: The reserved word **MarketPosition** is used to check the trade position for a bar on your chart. A value of* 1 *indicates that you are in a long position, a value of* -1 *indicates a short position, and* 0 *means that you are flat. Refer to the EasyLanguage Dictionary or search the TradeStation Help.*

Types of Orders

Up to this point, you've seen examples of buy, sellshort, and exit statements used in simple strategies that generate orders at the close of the bar being evaluated (which is the default for EasyLanguage). In this section you'll learn more about how EasyLanguage generates different types of orders, including stop and limit orders.

A *Buy* statement establishes a long position (regardless of the current position), a *SellShort* statement establishes a short position (regardless of the current position), and a *Sell* or *BuyToCover* statement liquidates an existing position (either long or short respectively).

One of the most important things to understand about TradeStation and EasyLanguage is that orders are always generated at the close of the current bar and "sent" either at the close of the current bar or on the next bar.

***Note**: In the context of paper trading, "sent" means you recieved that target price. In an actual trading contaxt, "sent" means that you will be prompted to place, modify or cancel the strategy generated order.*

This Bar on Close

Orders are evaluated at the close of a bar and, by default, the orders are placed using the bar's closing price. You can also add the phrase 'this bar on close' after a *Buy*, *SellShort*, *Sell*, or *BuyToCover* to do the same thing. For example:

```
if Condition1 then Buy this bar on Close ;
```

Next Bar at Market

A market order is placed at the price of the next available trade (the market price). For example, if you are charting daily bars, the following:

```
if Condition1 then SellShort next bar at market;
```

order to sell short would be "sent" at the open of the next day using the opening market price. Of course, there is no guarantee what the market price of the next trade might be, so an 'at market' order could result in a trade at a price higher or lower than desired.

Or higher (same as Buy-Stop and SellShort-Limit)

Instead of relying on the market price, you can write a statement that places an order if the next trade is at a price equal to the specified price *or Higher*. Depending on whether you want to buy or sell short, EasyLanguage automatically generates the proper stop or limit order based on your target price. For example,

```
if Condition1 then Buy next bar at 45 or higher ;
```

... generates a *buy stop order* for a price of 45 or greater, while:

```
if Condition1 then SellShort next bar at 68 or higher ;
```

... places a *sell short limit order* if the market trades at or above a price of 68.

It's important to understand that an *or Higher* order is "sent" only when the specified price condition is met during actual trading on the next bar. If the target price is not reached, the order is not "sent".

***Note**: Even though you've used **or Higher** to make your EasyLanguage easier to read and understand, when actually placing the order with your broker you'll need to use the appropriate Buy-Stop or Sell-Limit terminology.*

Or lower (same as Buy-Limit and SellShort-Stop)

You can also write a statement that places an order if the next trade is at a price equal to the specified price *or lower*. EasyLanguage automatically generates the proper stop or limit order based on your target price depending on whether you want to buy or sell short. For example,

```
if Condition1 then Buy next bar at 33 or lower ;
```

... generates a *buy limit order* for a price of 33 or less, while:

```
if Condition1 then SellShort next bar at 42 or lower ;
```

...places a *sell short stop order* if the market trades at or below a price of 42.

Remember, the *or Lower* order is "sent" only when the specified price condition is met during actual trading starting on the next bar. If the target price is not reached, the order is not "sent".

***Note**: Even though you've used **or Lower** to make your EasyLanguage easier to read and understand, when actually placing the order with your broker you'll need to use the appropriate Buy-Limit or Sell-Stop terminology.*

Points

Instead of needing to specify an absolute buy or sell short price when using *or Higher* and *or Lower*, you can place an order that will be "sent" only when the price on the next bar changes in the specified direction. A convenient way to do this is to add 1 point to a current bar value and let TradeStation calculate the next higher price based on the symbol's minimum movement. A *point* represents the smallest increment on the Price Scale for the symbol within TradeStation, while the *minimum movement* is the fewest number of *points* allowed for trading the symbol.

For example, the following buy order will be "sent" as soon as the price on the next bar exceeds the high of the current bar by any amount:

```
if Condition1 then
   Buy next bar at High + 1 point or higher ;
```

It’s important to note that the reserved word *point* in TradeStation refers to the decimal portion of a price and is typically set to 1/1000 (0.001) for stocks for maximum accuracy in calculations. For more information about price values, search the TradeStation Help.

Big Points

While the price for a stock symbol typically represents the value of each share in dollars, this is not necessarily true for other types of issues such as options and futures. For example, if we look at the S&P500 futures contract, a one integer (full point) change in price is valued at $250. The EasyLanguage reserved word *BigPointValue* is used to represent the number of dollars associated with a one integer (full point) change in a symbol’s price. Even though you don’t typically write entry orders in dollars, it’s useful to understand the relationship between a symbol’s price and the real value of the underlying asset. For example, the following buy order will be “sent” as soon as the price on the next bar meets or exceeds the high of the current bar plus one full integer or BigPoint:

```
if Condition1 then
   Buy next bar at High + 1 stop;
```

***Note**: The digit "1" by itself implies one full integer point (a BigPoint) where in the previous example "1 point" referred to a fractional point.*

Quantity

If you do nothing else, orders are placed for the number of *Fixed Units* as shown on the **General** tab of the **Format Strategy** dialog box. If you want to generate an order for a specified number of contracts or shares, you would add a number before the word ‘shares’ or ‘contracts’ in your order statement. For example, the following:

```
if Condition1 then SellShort 100 shares at 66 or higher;
```

places a sell short order for 100 shares if any trade occurs on the next bar for a price of 66 or greater, while this example:

```
if Condition1 then Buy 12 contracts next bar at market;
```

will place a buy order for 12 contracts at the market price of the next trade.

***Note**: In EasyLanguage, the reserved words **Shares** and **Contracts** are synonomous. They both refer to the number of <u>items</u> to purchase in a given trade. In other words, no attempt is made to distinguish between them when a buy order is generated so that you can use either word in a strategy to trade any type of symbol or market.*

Order Conditions

In summary, instead of placing a simple buy or sell short order at the close of the current bar, you can instruct TradeStation to place the order at the opening market price, or any price that is equal to or higher/lower than a specified price, by using additional phrases in your order statement. You can also specify how many shares/contracts to buy or sell short if you don’t want to use the default value specified for your strategy.

Exercises and Review

Review

Relational Operators are used in conditional expressions to compare values. The result of such a comparison is either *true* or *false*.

If…Then statements are used to perform an action when a simple or complex conditional expression is true.

Precedence of calculations is controlled through the use of parentheses. Operations enclosed within parentheses are calculated first, followed by multiplication or division, addition or subtraction, and relational comparisons.

Variables declarations must be performed before using a variable in a calculation except for EasyLanguage's built-in *ValueN* and *ConditionN* values.

EasyLanguage Dictionary can be used to paste any reserved word or function into a procedure.

Inputs are used to specify values in a procedure. Inputs are passed as parameters along with a function call or are set using the Inputs tab of an analysis technique.

Orders are processed at the close of the current bar. The four types of orders include: *Close* orders, *Market* orders, *Or Higher* orders, and *Or Lower* orders.

Market orders buy or sell short at the price of the next available trade. Market orders put in prior to the open of the next bar will be at the next bar's opening price.

Or Higher orders are placed as *Stop* orders when buying and *Limit* orders when selling short.

Or Lower orders are place as *Stop* orders when selling short and *Limit* orders when buying.

Close orders are placed at the close of the current bar (strategy default).

Exercises

(Answers are contained in Appendix A)

I. Mark the following either True or False (T or F).

1. Strategies are always complex EasyLanguage procedures.
2. Functions and variables return values.
3. The following is a numeric expression: Value1 + Value2 = Value3.
4. *If…Then* statements are used only in trading strategies.
5. Orders are always placed on the next bar.
6. A variable can be declared more than once in a procedure.
7. You can assign a value to any variable or input.

II. Identify each statement's type using the letters below:

A. Conditional statement

B. Declaration statement

C. Assignment Statement

1. Condition1 = High > High[1] ;
2. if Close < Close[1] then Buy next bar at market ;
3. variable: BuyPrice(0), SellPrice(0) ;
4. Value10 = (Close[1] + Close[2]) / 2;
5.
```
if Volume > Volume[1] then
    begin
    Value5 = Close[1] ;
    MyPrice = Close;
    end;
```
6. inputs: Price(0), Length(5);
7. if Close > Close[1] then Plot1(High, "UpClose");
8. SlowAvg = Average(Close[1], 9);

III. Write EasyLanguage statements for the following.

1. If today's high is greater than yesterday's close, buy 100 shares of a stock at tomorrow's open.

2. Buy as soon as the next bar's price is greater than today's high.

3. When the current bar closes up from the previous day's high, buy 25 shares at a price of $45 or higher.

4. When a stock's close is higher than yesterday's by 2 percent, you want to sell another 100 shares.

5. If you are in a long position and today's high is lower than yesterday's close, then you want to exit your position.

CHAPTER 3

More About Writing Trading Strategies

In this chapter you'll learn how to develop your ideas into complete strategies that combine multiple entry and exit conditions. You will also be introduced to the use of price data from more than one market. Finally, you'll learn more about customizing functions to take advantage of already developed ideas.

In addition to reading the topics and examples in this chapter, it is recommended that you complete the exercises and review questions at the end of the chapter.

In This Chapter

Defining Your Trading Rules

As an experienced trader, you probably already have ideas that you'd like to develop into trading strategies based on your observations, readings, and research. The power of TradeStation and EasyLanguage is that you are provided with a comprehensive set of tools for creating and optimizing strategies based on these ideas. Tools that let you 1) observe trend changes on your charts, 2) develop entry and exit orders that respond to identifiable trading patterns, and 3) historically test your ideas to increase the tradability of your strategy. In the end, you want your trading strategies to reliably follow your rules and be consistently profitable over time.

Set-up and entry

A key step in developing a trading strategy is to understand when and why you're entering the market, and how to do it. One popular method for doing this is called *set-up and entry*. The idea behind set-up and entry is to evaluate the market potential before actually placing an order. While it may not apply to all strategies, the set-up and entry concept is well proven and might help you uncover new possibilities for your trading rules.

Set-up

The set-up is used to identify conditions that must be present before making a decision to enter the market. The set-up looks at *when* you should think about entering a trade but does not actually place the trade. In essence, it's telling you to get ready to enter because the conditions could be right for a trade.

For example, in a trend-following strategy, a set-up would evaluate a change in market direction, such as when the fast moving average crosses over the slow moving average, or when the ADX indicates an increased trend strength. When your set-up proves to be true, you enter a kind of "entry mode" where you start looking for another set of conditions that will actually place the trade.

Although it's possible to place trades based solely on set-ups, this may not be in your best interest. For example, you wouldn't want to trade every time the market changes direction since that could result in many false trades that would cost you a lot of money in commissions alone.

Be aware that the actual comparison used to identify a set-up condition should be based on the type of strategy and market for which it's designed. For example, are you looking for a trend reversal, a swing in support or resistance, or perhaps a large gap in a volatile market? The set-up conditions needed for each of these markets might be quite different.

Entry

An entry represents the condition or conditions that will cause the actual trade to be placed once the rules for the set-up have been met. An entry condition confirms the direction of the set-up and determines *how* the order should be placed. In other words, once a *set-up* has placed you in "entry mode", the *entry* is the event that will actually pull the "trigger" and place the trade. It's quite common to use more than one entry condition with a given set-up. After all, you wouldn't want to miss the big move just because one specific entry condition wasn't true. For example, if your set-up gives you an uptrend signal and your only entry rule was to buy if a particular key reversal bar pattern occurs, you would miss other trading opportunities that might be equally valid entry conditions.

One of the important factors when developing entry rules is that, when all of your entries are combined, they should capture every price move for which they are designed, based on the matching set-up. In general, they should also confirm the direction indicated by the set-up before placing the trade.

Much like with set-ups, you could trade with just entry conditions, but using both together may provide a much stronger signal that can help eliminate the less profitable trades. Also, using the combination of set-up and entry lets you focus on defining your rules in a more objective way.

An example of a Buy order that uses a set-up and entry might be as follows:

```
if FastAvg crosses above SlowAvg and Close > Open then
   Buy next bar at market ;
```

where the fast moving average crossing over the slow moving average identifies the start of a trend (the set-up) and the **Close** > **Open** confirms the upward direction (the entry) before a buy order is placed for the next bar. In this example, the set-up is the first condition and the entry is the second.

Be aware that you can also use a conditional order as part of your entry strategy. For example, in the following:

```
if _CloseUps(3) then Buy next bar at High or higher ;
```

a buy stop (*or Higher)* order is generated after three consecutive bars close above their open (the set-up). The buy stop order is actually the entry condition since it states that the order can be triggered only if the next bar's price reaches or exceeds the current bar's **High**, thereby confirming the set-up. By the way, you'll learn how the *_CloseUps()* function works later in this chapter.

While there are limitless set-up and entry combinations, the important thing to remember is that your entry condition should confirm the direction of your set-up condition before generating a trade. Again, the set-up makes sure that the gun is pointing in the right direction and the entry pulls the trigger to actually fire at the target.

When to exit and why...

Just as important as knowing when to enter a position is knowing when to exit a position. Not only when, but why. For example, if the conditions that got you into the market are valid, then when those conditions change, it may be time to get out. Or, you may simply want to exit after being in the market for a fixed number of days. In any case, it's recommended that you think about how and why to exit so that you have a clear and repeatable set of rules.

For example, here are several possible exit scenarios:

- Conditions are changing and you want to take a profit.
- Your profit target was reached.
- You want to minimize a loss.
- The market is becoming too volatile.

Let's take a moment to expand on the thought process you might go through when creating an exit order to minimize a loss. At the moment you enter any trade, you should decide how much you're willing to give up if the trade doesn't go in your favor. In order to avoid losses greater than this amount, you might want to place a stop loss order with your broker. Later, you'll see how to write similar orders as part of a strategy.

Remember, planning your exits is <u>as important</u> as planning your entry strategy. For a strategy to be successful, you need to give careful thought to each exit condition, since having a clear exit plan can help protect you from indecision in trading situations where seconds can translate into dollars.

Money Management

Beyond simply entering and exiting the market, you should give some thought to how you want your trading strategy to help manage your money. In essence, how much will each trade cost and how much do you want to risk.

One way of doing this is to use multiple trades to increase your long or short position by adding or subtracting shares/contracts based on trend strength signals. This is called *pyramiding* and it allows you to add or subtract shares from an established position in separate trades. In this way, you limit how much you're going to risk on each trade while increasing your position as long as conditions are favorable or decreasing your holdings as conditions weaken.

Another money management technique is to use stop orders, which are designed to lock in gains while providing a safeguard against rapid market changes. This involves deciding how many dollars or percentage points you are willing to lose from your current position and adjusting the stop price with each trade so you can get out of your position if the market moves below/above that price. This is designed to allow you to keep the majority of your profits while quickly getting you out of the market when it goes against you.

Entry Orders

You've already worked with examples of simple entry orders in the previous chapter. In this section, you'll be learning about writing more sophisticated orders and how to use them to build complete trading strategies.

Entry orders are used to create a market position if none exists, or to reverse an existing position. Typically, a trading strategy should have at least one entry order and at least two exit orders (one for capturing profits and another for limiting losses).

Creating a position

In EasyLanguage, you use *buy* and *sell short* orders to establish an entry position or add to an existing position.

- A *Buy* statement creates a *long entry* position.
- A *SellShort* statement creates a *short entry* position.

If you are not in the market, a *Buy* order places you in a long position and a *SellShort* order places you in a short position.

Orders that include either a *Buy* or *SellShort* statement are considered entry orders.

Reversing a position

Buy and *SellShort* statements also are used to reverse your market position. For example, if you are already in the market with a long position, a *SellShort* statement actually closes out the position (goes flat) and then, as part of the same trade, places a sell short order. The reverse is true when you are short and use a *Buy* statement; EasyLanguage first closes out the short position and then goes long.

The important thing to remember about *Buy* and *SellShort* statements is that they always keep you in the market, either by reversing an existing position or by creating a new position. You can't be both long and short at the same time on a given chart.

Buy/SellShort

Up to this point, the sample entry orders that you created consisted of a single buy statement. It's quite possible to have multiple entry orders in a strategy that buy and sell short based on different rules. To keep things straight, you can name each buy or sell short statement by adding a name in parentheses after the reserved words *Buy* or *SellShort* as in the following examples:

```
if Close > High[1] then Buy ("Higher close") next bar at market;
```

```
if Volume > Volume[1] then
    Buy ("Volume up") 100 shares next barat market ;
```

```
if Close < Open then
    SellShort ("Down Bar") next bar at 48 or lower ;
```

Each of these examples generates an order based on a different condition and will display its order name next to the buy or sell short arrow on your chart, making it easy to see which order was generated.

More importantly, you can exit a particular trade by referencing its name in the appropriate *Sell* or *BuyToCover* statement. For example, the following buy statement:

```
if Close > High[1] then Buy ("CloseUp") next bar at Open ;
```

might have a matching exit order:

```
if Close < Low[1] then
    Sell from entry ("CloseUp") next bar at market ;
```

that exits the trade "CloseUp" but doesn't affect other open trades having a different name.

Also, if you specify a number of shares to buy or sell short, you can increase or decrease your position without completely closing it out.

Exit Orders

An exit order is the opposite of an entry order and is used to close out a market position. Unlike a traditional investor who might enter the market and stay, a trader typically needs to think about entering the market to catch a move while also planning how and when to exit. For example, it's quite common for a trader to buy into a trending market and exit later when a profit target is met, even though the initial trend is continuing. And the opposite is also true, where a trader might anticipate a movement in the market that fails to develop, and decide to exit with a limited loss.

In general, there are two basic reasons to exit a position. One is to take a profit and the other is to minimize a loss. It's recommended that you consider both reasons and use at least two exit conditions to accomplish these objectives in your strategies.

Closing a Position

In EasyLanguage, you use *Sell* and *BuyToCover* orders to close out a position (go flat).

- An *Sell* statement exits from a *long* position.
- An *BuyToCover* statement exits from a *short* position.

If you are not in the market, the *Sell* or *BuyToCover* statements are ignored by TradeStation.

Sell/BuyToCover

Just like their entry counterparts, the *Sell* and *BuyToCover* statements can place orders of any of the four basic types (see Types of Orders in Chapter 2). By default, orders are placed at the close of the current bar. For example:

```
if Condition1 then Sell this bar on Close ;
```

places an order to sell at the closing price of the current bar. In addition, if you don't specify a number of contracts or shares, a *Sell* or *BuyToCover* statement closes all trades for the matching type. In the above example, *Sell* closes out all long trades at the close of the bar.

Similarly, if you wanted to buy to cover a short position at the market price of the next trade, you would write:

```
if Condition1 then BuyToCover next bar at market ;
```

If you allow multiple open trades in your strategy, you can specify the number of shares or contracts to close by including “N shares” or “N contracts” after the exit order word, where N is the number of contracts to close for each open trade. Be aware that this will close N shares/contracts from each entry. For example, assume that your strategy had generated four previous buy orders with five contracts per order. In this case, the following exit statement:

```
if Condition1 then Sell 2 contracts this bar on Close;
```

would close out two contracts from each of the four long trades at the close of the current bar. This would leave you with three open contracts in each of the long trades.

If you want to close out just two contracts from the first open trade, you would write the following:

```
if Condition1 then Sell 2 contracts Total this bar on Close;
```

In this example, you would end up with three open contracts from the first long entry and five open contracts in the remaining two long trades.

Another way to use an exit order is to place a stop or limit exit order at the same time you establish the matching trade. For example, if your trade involves going long 200 shares on a stock worth $120 per share (a trade value of $24,000), you might decide that you only want to risk a maximum of 10% of the trade value (or $2400) if the market price decreases. When you place your *Buy* order, you would also place a *Sell Stop* order at the share price less 10% ($108 in this case). This way, if the stock price falls below your loss limit price, an order will be sent to exit the position. Here is what the EasyLanguage statements for both the *Buy* and *Sell* might look like:

```
inputs: OrderPrice(Close), RiskLoss(.10) ;

if High > Highest( High, 5 )[1] then
   begin
   Buy 200 shares on next bar at OrderPrice stop ;
   Sell on next bar at OrderPrice * ( 1 - RiskLoss ) or lower ;
   end ;
```

Notice the use of inputs to make the strategy more flexible by allowing you to set the price and risk value at the time you apply the strategy to a chart.

Multi-data Strategies

A multi-data strategy makes use of TradeStation’s powerful ability to reference price and trade information from more than one data stream. For example, let’s say that you want to compare a stock’s price to the overall exchange index before making a buy or sell decision. If you add the symbol for a stock and the symbol for its exchange index to your chart, you can refer to either data stream from EasyLanguage. Typically, the charted stock will be **Data1** and the index will be **Data2**. In the example below:

```
Condition1 = Close of Data1 > Close[1] of Data1 ;
Condition2 = Low of Data2 < Low[1] of Data2 and
             Close of Data2 > Close[1] of Data2 ;
if Condition1 and Condition2 then
   Buy next bar at market ;
```

Condition1 is true if the current bar's close is greater than the previous bar for the main symbol (Data1) and Condition2 is true if a key reversal up occurs in the index (Data2). In other words, if the stock price is up and the index is reversing up from the previous bar, then buy.

By adding the phrase "of DataN" after a function, you can make it refer to prices from the specified data stream, such as:

```
Value1 = Average( Close, 10 ) of Data2 ;
```

to get the 10-bar moving average of the closing price from the data stream applied as *Data2*.

You can just as easily compare one stock or commodity against another, compare market indexes, or look at the relationship between groups of issues. EasyLanguage allows you to reference up to 50 data streams on a single chart, including the main data stream (**Data1)** and 49 additional streams (**Data2** through **Data50** respectively). However, one word of caution. You must be sure that you properly assign the correct symbol to the desired **DataN** channel.

Custom Functions

Although there are dozens of functions built into TradeStation, you may find a need to change a function or create your own. Once you understand how functions operate, you'll discover that it's also easy to make your own custom functions based on those included in the EasyLanguage Dictionary. You can easily copy the contents of these functions and make your own variations that can be used in any strategy, indicator, or analysis technique.

Let's say that you're developing an entry order based on the close being greater than the open for the previous 3 bars. One way to do this might be to declare a true/false variable and write a multiple condition *If…Then* to test each bar like this:

```
variable: CloseUp(False) ;
CloseUp = Close > Open ;
if CloseUp[1] and CloseUp[2] and CloseUp[3] then
   Buy next bar at market ;
```

But what if you wanted to perform the same test over the past 5 bars, or the last 10? The *If…Then* statement would get much too long and hard to read. The solution is to write a custom function that is true when a condition of your choosing occurs on each of the last N bars. This is easier than you might think. The EasyLanguage Dictionary already includes a function called *CountIf*(condition,length) that counts the number of times a condition occurs over a specified number of bars. For example, *CountIf(*Close>Open,10) would return a value of 3 if the condition *Close>Open* happened 3 times during the last 10 bars.

So let's go back to the previous example. We'll write a function, based on *CountIf*, that is true when a specified condition occurs on each of the previous N bars. First, we need to write a comparison that tests whether the condition *Close>Open* occurs 3 times during the last 3 bars. The EasyLanguage for this would be:

```
if CountIf( Close > Open, 3 ) = 3 then ACTION ;
```

When a condition occurs three times over the past three bars, it is the same as saying that the condition occurred on each of the last 3 bars (the current bar and the previous two).

But, remember, we want to create a function that tests for the occurrence of our condition for any number of previous bars. In the previous chapter, you learned about the idea of using inputs to pre-set values before running a procedure. This is especially important when writing a function. The parameters included after the function's name become inputs that are used in the function's calculations and comparisons. Inputs within a function do not have any initial value, but you must indicate the type of value that each input represents (numeric, true/false, string). The function will require a numeric length for its input like this:

```
inputs: Length(Numeric) ;
```

where *Length* is the number of consecutive occurrences that will make the function true. Note that the data type (numeric) of an input and its matching parameter must be the same.

Now, when you replace the length value in the previous comparison with the new input, you get:

```
 if CountIf( Close > Open, Length ) = Length then ACTION ;
```

where input *Length* replaces the number **3** in both the *CountIf* parameter and to the right of the equal sign.

Now, create a new function named *_CloseUps*. ***Hint****: In the* ***New Function*** *dialog box, select* TrueFalse *under* Return Type. Type the following EasyLanguage statements:

```
input: Length(Numeric) ;
if CountIf( Close > Open, Length ) = Length then
   _CloseUps = True
Else
   _CloseUps = False ;
```

Example 3-1. Function *_CloseUps*.

Verify the function. Notice that, based on the condition, *True* or *False* is assigned to the name of the function (*_CloseUps* in this case). This becomes the value of the function and is always set by assigning an expression to the function's name.

Now, create a new strategy named *_CloseOpen* and enter the following EasyLanguage statement:

```
if _CloseUps(3) then Buy next bar at market;
```

Verify the strategy. This strategy places a buy order when 3 consecutive bars close higher than they open. Remember, you could also replace the value "3" with an Input to make your strategy more flexible.

Finally, apply both the *_CloseOpen* and *TimeExit* (*Bars)* strategies to your sample chart and observe the **Buy** orders following each three-bar pattern where the close is greater than the open (Figure 3-1).

The custom *_CloseUps* function you created for the *_CloseOpen* entry order can also be used in any analysis technique where you need to test whether the condition occurs over the previous N bars. Once it's developed and proven, a function is a powerful tool that may help make your EasyLanguage expressions easier to read and less prone to errors.

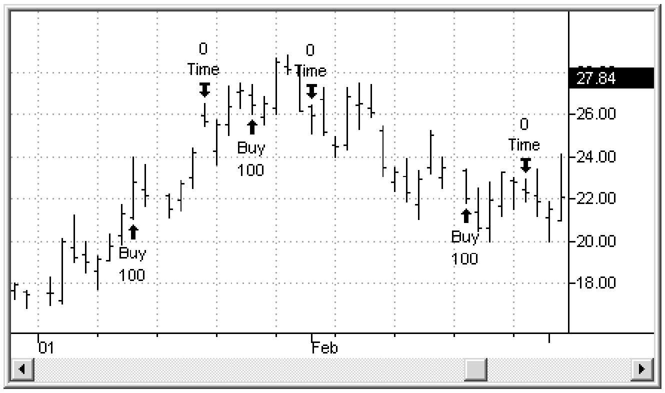


Figure 3-1. Strategy *_CloseOpen.*

While creating your own functions is not difficult, it may not be necessary for you to ever write one because the standard EasyLanguage Dictionary already includes a wide variety of usable trading functions. To learn more about EasyLanguage and functions, it may be useful to go into the PowerEditor and look at the EasyLanguage instructions for some of the built-in functions. Also, additional reference material about functions can be found by searching the TradeStation Help.

Exercises and Review

Review

Setup and Entry is a trading methodology that is based on the idea of using a setup to establish an “entry mode” before actually using an entry to “trigger” the placement of an order.

An **Entry Order** is a TradeStation procedure that is used to establish a long or short position using the EasyLanguage words *Buy* and *SellShort*.

An **Exit Order** is a TradeStation procedure that closes out an open position. The reserved word *Sell* closes out a long position and *BuyToCover* closes out a short position.

Stops are used to generate exit orders (stop or limit orders) that are designed to minimize risk or capture profits when prices move. Risk avoidance stops are sometimes referred to as *Protective Stops*.

Multi-data Strategies use more than one data stream for comparison and calculations.

Functions allow you to easily reference commonly used calculations. A large number of built-in functions are in the EasyLanguage Dictionary. Users can also create their own custom functions.

Exercises

(Answers are contained in Appendix A)

I. Mark the following either True or False (T or F).

1. Set-up and Entry are two standard signals in TradeStation.
2. A Buy statement enters a long position.
3. A SellShort statement exits the market.
4. The phrase *Sell 2 Contracts* closes out 2 contracts from each long trade.
5. A multi-data strategy looks at data from more than one chart at a time.
6. Multi-data strategies cannot place a buy order.
7. A trailing stop is used to exit from a trade after a specified number of days.
8. Every strategy must have an entry and an exit.
9. The default value for inputs in a function is different than for a study.

II. Identify each order type using the letters below:

A. Enter Long Position	C. Close Out Long Position
B. Enter Short Position	D. Close Out Short Position

1. BuyToCover this bar at Close ;
2. if Close > High[1] then Buy next bar at market ;
3. if Volume < Volume[1] then SellShort this bar on Close;
4. if Average(Price, FastLen) crosses over Average(Price, SlowLen) then
   Buy this bar at Close ;
5. if MarketPosition <> 1 then Sell next bar at PBase * (1 - Pcnt) stop ;
6. if Close > Close[1] then Buy next bar at 100 or lower ;
7. if Close > Open then

   ```
   begin
   Buy next bar at market ;
   Sell next bar at Close * .90 stop ;
   end ;
   ```
8. if _CloseUps(3) then Buy 50 shares next bar at market ;

CHAPTER 4

Creating Indicators and Studies

In this chapter, you'll learn how to use EasyLanguage to develop indicators and studies. You will be introduced to the plot statement and to the use of charting for data analysis. Finally, you'll gain additional practice in translating trading ideas into EasyLanguage instructions.

The material and examples in this chapter cover the data analysis side of developing your trading ideas. To gain additional experience, it's a good idea to complete the exercises and review questions.

In This Chapter

Understanding the Flow

An important skill in developing trading ideas is the ability to visually identify trends and patterns on a chart. The more you can identify relationships between prices and bars, the easier it becomes to create your own trading rules and the EasyLanguage conditions used to evaluate them.

In addition to letting you develop trading strategies and functions, EasyLanguage also allows you to create other types of analysis techniques, such as indicators and studies. By plotting graph lines, text, and other symbols on a chart, you can use indicators and studies to help you see patterns that reveal market activity and trends.

Indicators

An indicator is the general name for an EasyLanguage analysis technique that calculates and displays values based on price data changes for each bar.

For example, an indicator might draw reference lines or symbols on top of a chart, such as the *Mov Avg 2 Lines* indicator (Figure 4-1) which plots lines for both the fast and slow moving average.

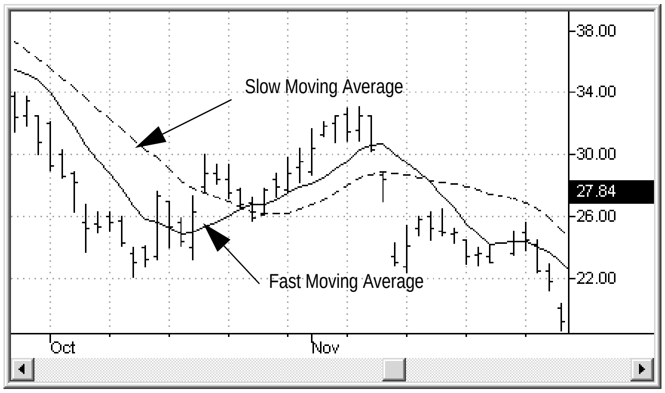


Figure 4-1. Indicator *Mov Avg 2 Lines*.

Another style of indicator plots information beneath your bar chart, such as the *Volume* indicator (Figure 4-2) which shows a histogram of the trade volume for each bar.

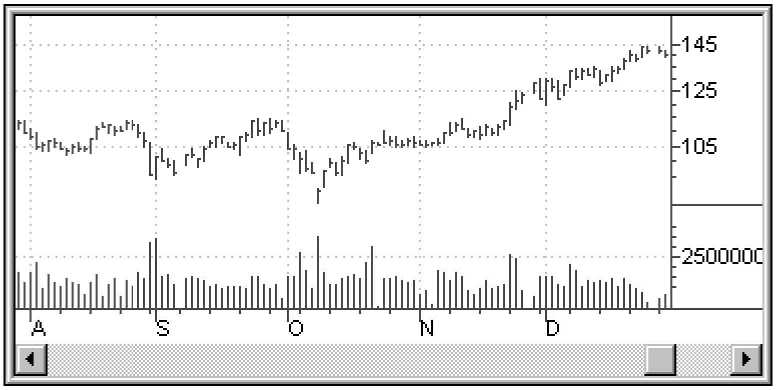


Figure 4-2. Indicator *Volume* histogram.

Studies

A study is a special type of analysis technique that plots information on a chart in a specific format. The different types of studies included with TradeStation are: ShowMe, PaintBar, ActivityBar, and ProbabilityMap. Each has a distinctive appearance and purpose.

ShowMe

A ShowMe study places a marker above or below any bar that matches the conditions stated in the ShowMe procedure. Unlike a typical indicator that draws a continuous line on or below a set of price bars, a ShowMe only marks the bars matching a specific condition. For example, you might use a ShowMe to mark every inside bar, as in the following example (Figure 4-3):

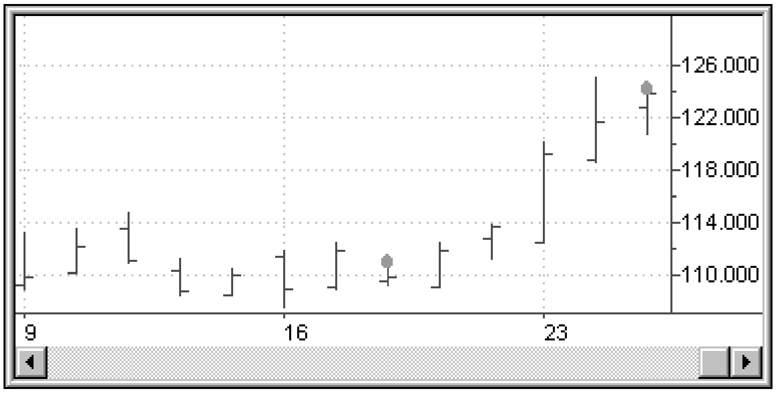


Figure 4-3. ShowMe study *Inside Bar*.

An inside bar is one that could fit 'inside' the previous bar, or where the bar's **High** is less than the previous **High** and the bar's **Low** is greater than the previous **Low**. Notice how the circle above each bar calls attention to this condition without the cluttered appearance associated with some indicators.

PaintBar

A PaintBar study changes the color of bars that match a stated condition. For example, the *Momentum Increasing* PaintBar (Figure 4-4) colors each bar where the momentum is increasing. Although most PaintBar studies color the entire length of a bar, it's possible to color only a selected portion of a bar.

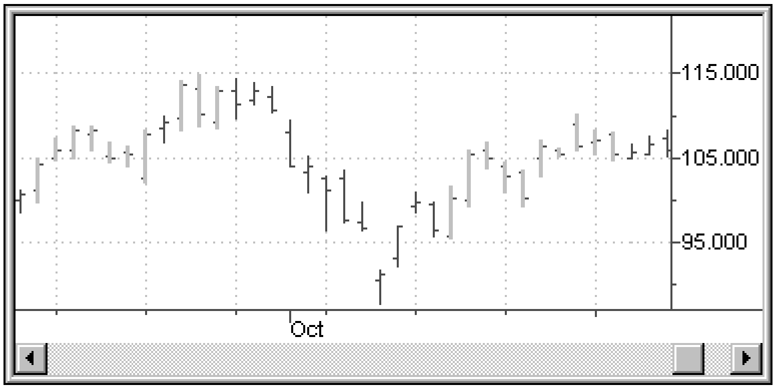


Figure 4-4. PaintBar study *Momentum Increasing*.

ActivityBar

An ActivityBar study is designed to let you actually look at the trades that make up a bar by extending colored or shaded “activity bars” to either side of the vertical price bar. For example, in the ActivityBar study named *Price Distribution* (Figure 4-5), you can see how prices developed during the trading period of each bar.

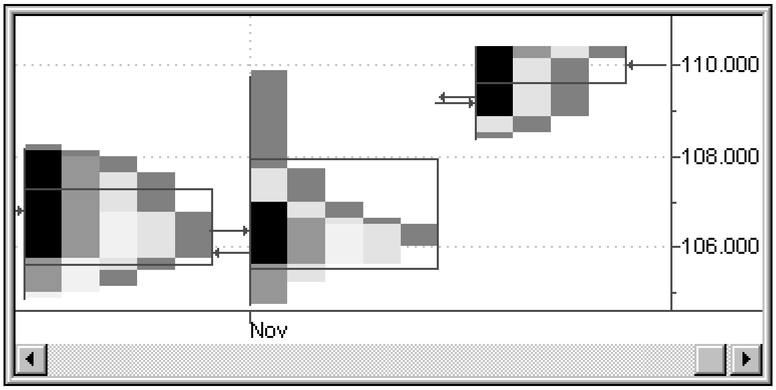


Figure 4-5. ActivityBar study *Price Distribution*.

ProbabilityMap

The ProbabilityMap study (Figure 4-6) lets us view potential price changes using probability calculations derived from the symbol’s recent trading history. For example, you can extend a chart into the future to get an idea of the direction of potential price movement.

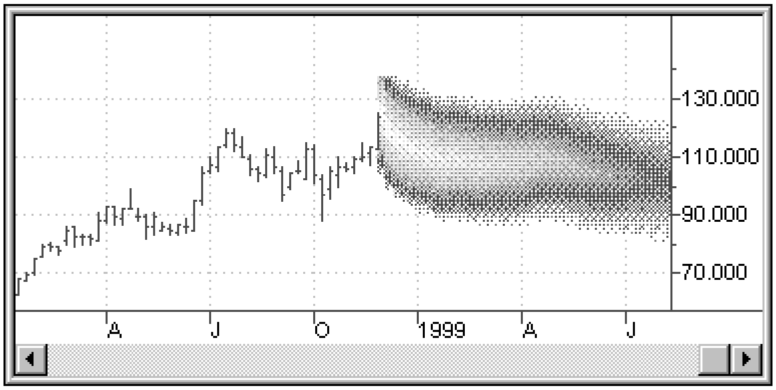


Figure 4-6. ProbabilityMap study.

Reading Data

Just like with trading strategies, an indicator or study looks at the price data for each bar on the chart, starting from the left and moving to the right (refer to Chapter 1, Overview). In EasyLanguage, the *current bar* is the name given to the bar that your procedure is currently evaluating. On each bar, EasyLanguage reads the current bar's price data and typically compares it with data from previous bars.

In EasyLanguage, the closing price for the current bar is written as **Close** while the same price from the previous bar (one bar ago) is **Close**[1]. For example, if you wanted to perform some action when the current bar's close is greater than the high of the previous bar, you would write:

```
          if Close > High[1] then ACTION ;
```

which reads "if the closing price of the current bar is greater than the high price of one bar ago, then perform a designated action."

For indicators and studies, the action is to plot a line or symbol at some location on a chart or grid. The following sections describe the differences between these analysis techniques and their plotting formats.

Your First Indicator

Writing indicators involves many of the EasyLanguage skills that you already used when creating trading strategies. Unlike strategies, indicators do not place orders, but they do have the ability to display multiple plots on your chart based on price calculations and comparisons.

Plot statement

The plot statement is used in indicators and studies to draw lines and text on a chart. The simplest form consists of a value to be plotted. In EasyLanguage, the plot statement looks like this:

```
   Plot1(High) ;
         Value
```

The *Value* parameter is plotted using your choice of continuous lines, histogram bars, or other symbols. The appearance of your plot (color, thickness, etc.) can be changed using the settings under the **Style** and **Color** tabs on the **Format Indicator** dialog box. A plot statement can optionally include a *Text Field* that helps identify the plot on the **Style** and/ or **Color** tabs.

```
   Plot1( High, "My Plot Name" ) ;
          Value     Text Field  (optional)
```

You can have as many as four plot statements (Plot1 through Plot4) in your procedure.

Now, let's write a simple indicator that plots a line between the closing price of each bar. Create a new **Indicator** named _*Close*, then type the following EasyLanguage instruction:

```
Plot1(Close) ;
```

Example 4-1. Indicator _*Close*.

Verify your indicator.

Switch back to your sample chart and use the **Insert – Indicator** menu sequence to select the _*Close* indicator you just created, then click **OK**. From the **Format Indicator** dialog box, click the **Scaling** tab and make sure that the **Scale Type** is set to *Same As Symbol.* This instructs TradeStation to plot the indicator on top of the bars on your chart. Click **OK** to apply the indicator to your chart, and observe the line drawn between the close of each bar. It should look something like this:

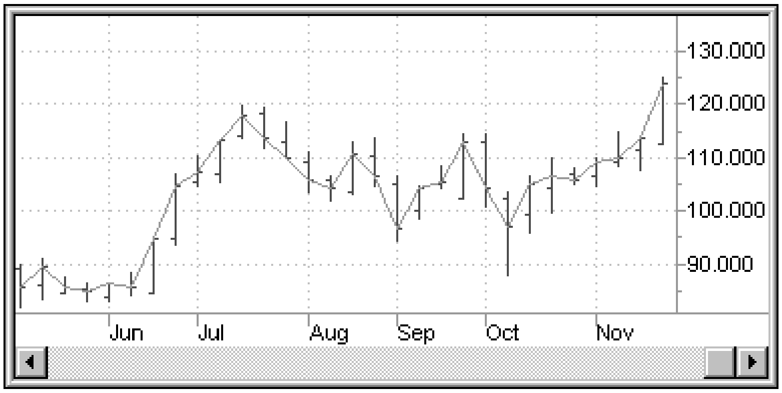


Figure 4-7. Simple Indicator _*Close.*

Style and Scaling

The *style* options control the visual characteristics of an analysis technique (color, line type and style, weight, etc.) while the *scaling* options determine where the analysis technique will be plotted relative to the primary symbol chart (overlaid on the bars, beneath the bars, etc.). You can change the style and scaling of your analysis technique at the time you apply it to a chart, or you can set the default properties of your analysis technique as you create it.

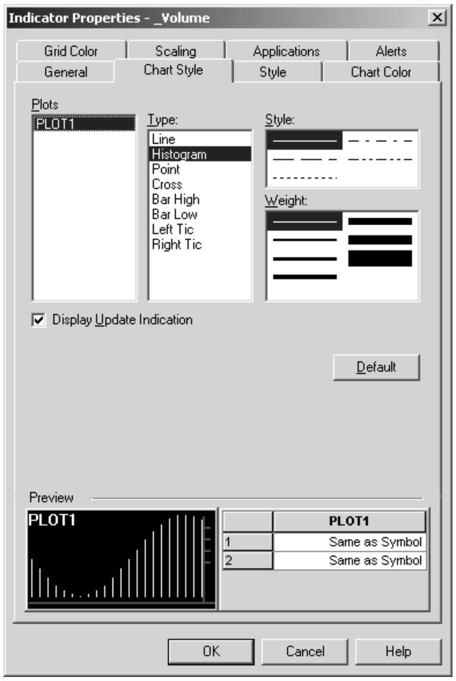
Let's try creating another indicator and learn how to change the default style and scaling properties. Create a new **Indicator** named _*Volume*, and type in the following EasyLanguage statement:

```
Plot1(Volume) ;
```

Example 4-2. Indicator _*Volume*.

Verify your new _*Volume* indicator.

While the PowerEditor window is still active, use the **Format - Properties** menu sequence to display the **Indicator Properties** dialog box. Click the **Chart Style** tab and select *Histogram* as the line **Type**. Observe the change in appearance of the sample plot at the bottom of the dialog box (see Figure 4-8). Also, on the **Scaling** tab, make sure that the **Scale Type** is set to *Screen* so that your plot appears in a subgraph beneath your bar chart. For reference, look at the options on the other tabs to become familiar with the default appearance of your analysis technique.


Indicator Properties - _Volume

Plots: PLOT1

Type: Histogram

☑ Display Update Indication

	PLOT1
1	Same as Symbol
2	Same as Symbol

Figure 4-8. Indicator Properties.

After you're done setting the properties, click **OK**. Switch back to your sample chart and apply it the indicator _*Volume*. The indicator should appear beneath your chart as shown in Figure 4-9.

If you didn't remove the previous _*Close* indicator, it may still be on your chart as well.

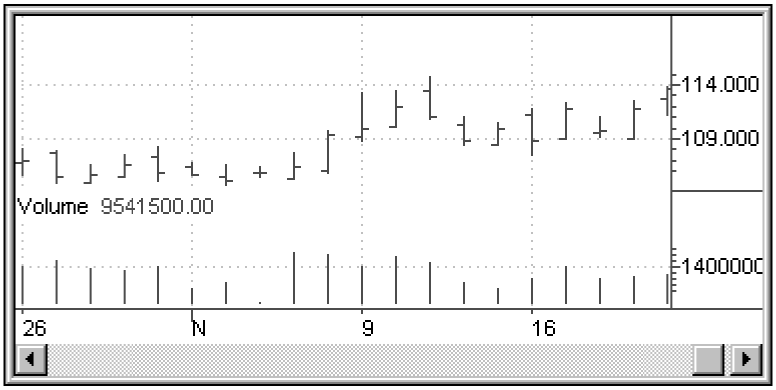


Figure 4-9. Indicator named _*Volume.*

But don't worry, you can combine multiple indicators on a chart without any difficulty. By the way, to remove an indicator or study from your chart, click on it (observe the square selection markers) and then press the DELETE key.

Writing Studies and Alerts

In addition to writing your own indicators, you can also create custom ShowMe, PaintBar, ActivityBar, and ProbabilityMap studies. Even though they all plot information on a chart, they each do it in a different way. For example, indicators typically plot the same type of information from bar to bar, such as a continuous moving average line or a histogram showing each bar's volume. On the other hand, a ShowMe or PaintBar study commonly marks selected bars based on the result of a conditional expression. In this way, studies are similar to trading strategies, except that studies do not place orders. Finally, ActivityBar and ProbabilityMap studies make use of additional plotting functions beyond the basic plot statement and require a more advanced understanding of EasyLanguage.

Writing a ShowMe Study

A ShowMe study places a marker on a bar based on a conditional expression. It is common to use ShowMe studies to visually identify key price events in preparation for using the idea in a trading strategy. For example, you might use a ShowMe to mark each bar that is preceded by a series of up closes for a specified number of days. Or, a ShowMe could mark each inside bar, where the high is less than the previous high and the low is greater than the previous low.

By default, the plot statement for a ShowMe study draws a marker at the specified price (typically the bar's high for upward movement and the low for downward movement). You can change the plot style and color for a ShowMe using tabbed items in the **Properties** dialog box.

Now, let's create a ShowMe study that marks a bar that is preceded by three bars that closed higher than they opened. Create a new **ShowMe** file named *_3UpCloses*. Type the following EasyLanguage instructions:

```
Variable: UpClose(False);

UpClose = Close > Open ;
if UpClose[1] and UpClose[2] and UpClose[3] then
   Plot1(High) ;
```

Example 4-3. ShowMe study *_3UpCloses*.

Verify the ShowMe study. Go to your sample TradeStation chart and apply the ShowMe named *_3UpCloses* to your data. Observe that each marked bar follows three up closes.

You could also have written the above ShowMe using the *_CloseUps* function that you created in the previous chapter. The following EasyLanguage statement does exactly the same thing as the several statements listed above:

```
if _CloseUps(3)[1] then Plot1( High, "3UpCloses" ) ;
```

***Note**: Look at the use of the "[1]" (of 1 bar ago) after the _CloseUps function. This instructs the function to test for 3 consecutive occurrences of **Close**>**Open** starting with the previous bar. If you eliminate the "[1]" from the statement, the test would include the current bar and the previous 2 bars (still a total of 3 consecutive bars).*

Although both of the above examples are valid, the *_CloseUps* variation gives you more flexibility since you can easily use inputs for the function's parameters and have a ShowMe that can look for a number of different conditions. The resulting EasyLanguage statements for the new ShowMe named *_ShowCloseUp* would look like:

```
inputs: Length(3) ;
if _CloseUps(Length)[1] then
   Plot1( High, "_ShowCloseUp" ) ;
```

Example 4-4. ShowMe study *_ShowCloseUp*.

Create a ShowMe named *_ShowCloseUp* using the above statements. Apply it to a chart and observe that it produces exactly the same plot as the *_3UpCloses* example. However, the new ShowMe lets you change the length when it's applied to a chart, making it much more flexible.

Writing a PaintBar Study

A PaintBar study changes the appearance of a bar based on a conditional expression. PaintBar studies make is easy to visually identify a series of bars that share a common characteristic.

Instead of using a single plot statement to mark a bar, the PaintBar study uses a pair of plot statements to indicate the color or style change on a bar. The first plot specifies where to start painting the bar and the second plot specifies where to stop painting the bar. For example, the following pair of plots:

```
if Condition1 then
   Begin
   Plot1( High, "Start_High" ) ;
   Plot2( Low, "End_Low" ) ;
   End ;
```

paints the entire length of each bar (from the *High* price to the *Low* price) where *Condition1* is true. You could just as easily paint only a part of the bar, from the close to the open, for instance. You can change the plot style and color for a PaintBar using tabbed items in the **Properties** dialog box.

Now, let’s create a PaintBar study that marks a series of bars that are trending up based on the fast moving average being greater than the slow moving average. Create a new **PaintBar** file named *_BullAvgs*. Type the following EasyLanguage instructions:

```
variables: FastAvg(0), SlowAvg(0) ;

FastAvg = Average( Close, 9 ) ;
SlowAvg = Average( Close, 18 ) ;

if FastAvg > SlowAvg then begin
    Plot1( High, "BarHigh" ) ;
    Plot2( Low, "BarLow" ) ;
end ;
```

Example 4-5. PaintBar study *_BullAvgs*.

Verify the PaintBar study. Go to your sample TradeStation chart and apply the PaintBar named *_BullAvgs* to your data. Observe the marked bars. Now, apply the *Mov Avg 2*

Lines indicator to your chart and notice that the PaintBar study has marked all bars that are part of the upward trending cycle (see Figure 4-10).

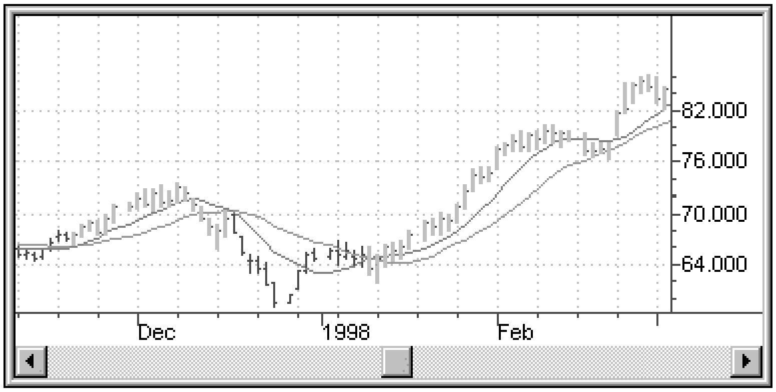


Figure 4-10. PaintBar study *_BullAvgs* along with the *Mov Avg 2 lines* indicator.

In the previous EasyLanguage example, notice the use of the block reserved words *Begin…End* as part of the *If…Then* statement. Remember, this allows EasyLanguage to perform more than one action if the condition is true.

Writing Alerts

An alert is another type of action that an indicator or study can perform. Instead of drawing a line or symbol on a chart, an alert displays an ‘alert’ message box on your monitor and sends an alert summary message to the Message Center. For example, when a pair of moving average lines cross, an alert could be generated informing you of the cross over condition.

Alerts are triggered based on the last bar in the chart. That means that an alert message will be produced whenever a specific alert condition is true for the last bar in the chart. In the following example:

```
if Close > High[1] then Alert ;
```

an alert message will appear whenever the close of the last bar is greater than the high of the previous bar. However, if the last bar closes lower than the previous bar’s high, no alert is generated even if the condition might have been true on previous bars since alerts are only valid for the last complete bar on a chart.

When writing and using alerts, you need to be sure that the **Enable Alert** box is checked on the **Alerts** tab of the **Format [Analysis Technique]** dialog box when you apply an indicator or a study to your chart. You can also set this property when you create the indicator/study by changing the **Properties** from the PowerEditor.

Using Inputs

As you create your own indicators and studies, you should think about the idea of using inputs for values that you might want to change when you apply the indicator and study to your chart. For example, with an indicator that uses a pair of moving averages, you could use inputs to set the number of bars on which to calculate both the fast and the slow averages. This increases the flexibility of your analysis techniques by letting the user set the input values when applying them to a chart.

Let's make an indicator based on one that you created earlier. Go back to the indicator you created earlier named *_Volume*. You're going to add a second plot that shows the moving average for the volume over the past N bars. You'll be using the *Average* function from the EasyLanguage Dictionary. In addition, you'll use an input to set the number of days on which to compute the average. Change your EasyLanguage statements to read:

```
inputs: Length(10) ;
variable: AvgVol(0) ;

AvgVol = Average( Volume, Length ) ;

Plot1( Volume, "VolumeBars" ) ;
Plot2( AvgVol, "AvgVol" ) ;
```

Example 4-6. Indicator *_VolumeAvg*.

Use the **File - Save As** menu sequence and give your new indicator the name *_VolumeAvg*. Verify the indicator. Switch to your sample chart and insert the new indicator. The new indicator plots a histogram of the volume and also includes a plot of the 10-day average volume (see Figure 4-11).

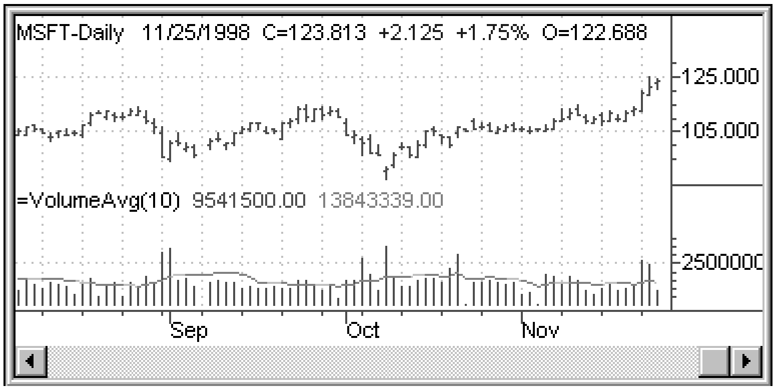

MSFT-Daily 11/25/1998 C=123.813 +2.125 +1.75% O=122.688
125.000
105.000
=VolumeAvg(10) 9541500.00 13843339.00
2500000
Sep
Oct
Nov

Figure 4-11. Indicator *_VolumeAvg.*

Since you used an input for the length of the average in the *_VolumeAvg* indicator, you can change the value of *Length* from the **Inputs** tab on the **Format Indicator** dialog box whenever you insert the indicator.

Exercises and Review

Review

Analysis Technique is an EasyLanguage procedure used to analyze price data. All indicators, studies, and trading strategies are considered analysis techniques.

ShowMe is a particular type of study that places a marker above or below a bar that matches one or more conditions. ShowMe studies are best at identifying occurrences such as a key reversal or a moving average crossover.

PaintBar is a type of study that changes the color or style of bars matching a set of conditions. PaintBar studies are best at identifying modes such as a group of bars that are part of an uptrend.

ActivityBar is an EasyLanguage study type that builds a set of secondary bars to the right or left of a bar so that you can see trading activity within a bar.

ProbabilityMap is a type of study that allows you to observe probable price changes based on recent history.

The **Plot** statement draws lines and symbols on a chart at designated price points. It is used in indicators and studies.

An **Alert** statement produces an on-screen message when a particular price event occurs and places a corresponding entry in the tracking center.

Exercises

(Answers are contained in Appendix A)

I. Mark the following either True or False (T or F).

1. An indicator is not an analysis technique.
2. A ShowMe study changes the color of a bar based on a condition.
3. All indicators and studies must include a plot statement.
4. Alerts occur when a condition is true on any bar.
5. A PaintBar study uses at least two plot statements to draw on a chart.
6. Line styles and scaling must be set at the time an analysis technique is applied to a chart.

CHAPTER 5

More About EasyLanguage

In this chapter, you'll learn more about the power and flexibility of EasyLanguage. You'll be introduced to additional terms and data types that increase the sophistication of your strategies and analysis techniques.

Many of the features described in this section are for advanced users but should be of interest to all. It is recommended that you complete the exercises and review questions at the end of the chapter to get the most out of this material.

In This Chapter

Advanced Grammar and Data Types

In the previous chapters you learned the basic vocabulary and structure of EasyLanguage. Now it's time to take a quick look at some advanced features.

Qualifiers

Whenever you use a price value, such as **Close** or **High**, in a calculation or comparison it is assumed that you are referring to the prices associated with the primary data stream, or *Data1*. This default operation of EasyLanguage makes simple instructions easier to read and understand. However, if you are working with multi-data charts and analysis techniques, you can also refer to prices from another data stream by using the data qualifier "*of xxx*" after each price value. For example, you might want to place a buy order if the closing price from two different data streams has increased from the previous bar:

```
Condition1 = Close of Data1 > Close[1] of Data1 ;
Condition2 = Close of Data2 > Close[1] of Data2 ;

if Condition1 and Condition2 then
   Buy next bar at market ;
```

Remember, each data stream can reference prices for each bar in the stream for a total of *MaxBarsBack*.

In addition to the qualifiers for Data1 through Data50 (*of Data1…of Data50*) there is a separate qualifier for ActivityBar data (*of ActivityData*). For more information on multiple data streams, see the TradeStation Help.

Text Values

In addition to the two basic data types (numeric and true/false), EasyLanguage also has limited support for text values (also known as a text *string* in computer jargon). A text string is a series of characters within quotation marks as follows:

```
variables: MyString1(""), MyString2("") ;

MyString1 = "A series of characters " ;
MyString2 = "or words" ;
```

It's important to note that you must initialize a variable using a string (such as a pair of quotation marks) before it can be assigned a text value.

EasyLanguage allows you to combine text strings for use within *Print* statements or in messages by using the plus (+) operator. In this example:

```
MyString3 = MyString1 + MyString2 ;
```

the variable *MyString3* will contain a single text string "A series of characters or words" made up from the two phrases in *MyString1* and *MyString2*.

The ability to build text strings is useful with the *Print* or *Commentary* statements. Sending text to the Print Log or to the Commentary window helps when debugging (troubleshooting) your indicators, studies, and strategies. It allows you to see actual written values that can help track down errors in your conditions or calculations.

For example, you could create a text string and use the *Print* statement to send it to the Print Log along with other price information, such as the symbol name, using a Print statement like this:

```
variable: MyText("");
MyText = "My stock symbol is: "  ;
Print( MyText, GetSymbolName ) ;
```

The resulting Print Log entry would read:

```
My stock symbol is: MSFT
```

You can also send text to a file or the printer. For more information, see the TradeStation Help.

Advanced Structures

If...Then...Else

Since a standard *If…Then* statement performs an action only when the condition is true, there are times when you might want to perform an alternate action when the condition is false. In that case, you would use the *If…Then…Else* statement which, in English, reads "**if** a condition is true, **then** perform an action, or **else** do a different action."

For example, you might want to buy if the current bar closes up and sell if it doesn't, as in this strategy:

```
if Close > Close[1] then
   Buy 20 shares next bar at market
else
   Sell 10 shares next bar at market ;
```

You can perform multiple actions after either the *Then* or *Else* portion of the statement by using the block *Begin…End* words before and after your action instructions. An example of the block form of the *If…Then…Else* would be:

```
if Close > High[1] then
   begin
   Value1 = 10 ;
   Condition1 = true ;
   end
else
```

```
begin
Value1 = 20 ;
Condition1 = false ;
end ;
```

Notice that there are no semicolons after the words *Begin* and no semicolon after the first *End.* That's because these are considered part of the complete *If…Then…Else* statement. If you put a semicolon in the wrong place, you will get a verification error message as a reminder.

Loops

In trading, you may want to perform operations on a range of values, such as calculating the average price for the last 10 bars. In fact, that's exactly what many EasyLanguage functions do by 'looping' through a series of repetitive calculations. There are two EasyLanguage statements that can be used for this purpose. The first of these is the *For…Begin* statement that loops for a specified number of times, and the second is the *While…Begin* statement that loops as long as a condition remains true.

For...Begin

The purpose of a *For…Begin* loop is to perform a set of actions a specified number of times. A counter variable is used to count the number of steps through the loop based on the starting and ending values that appear after the equal sign. The EasyLanguage statements that appear between the *Begin…End* reserved words are processed each time through the loop.

Look at the following general example:

```
for Value1 = 0 To 5
   begin
   ACTIONS
   end ;
```

The first time through the *For* loop, the value of the counter variable (*Value1*) is set to 0 and the statements (ACTIONS) between *Begin* and *End* are processed. When the bottom of the loop is reached, EasyLanguage moves back to the top of the loop, increments the value of the counter variable (*Value1*), and performs the ACTIONS again. In the above example, the loop would be executed 6 times using counter values of 0, 1, 2, 3, 4, 5.

To make a backward counting *For...Begin* loop, replace the word *To* with the word *DownTo* as in the following example:

```
variable: MyValue(0) ;
for MyValue = 5 DownTo 1
   begin
   ACTIONS ;
   end ;
```

the ACTIONS in this loop will be executed 5 times with *MyValue* containing values starting with 5 and ending at 1.

The counter variable may be either a built-in variable (Value1 through Value99) or any user declared numeric variable.

While…Begin

The *While…Begin* loop is used to execute a block of statements an indefinite number of times. If the condition following the word *While* is true, the statements between *Begin…End* are processed. When the *End* is reached, EasyLanguage returns to the top of the loop and tests the condition again. The loop repeats as long as the condition following the word *While* remains true.

It's important to understand that a *While…Begin* loop has the potential of running indefinitely. As long as the condition is true, the statements in the loop will be processed. Therefore, it's important that you use a condition that changes from true to false to avoid getting an application error. Also, be aware that if the condition starts out false, the statements in the loop will never be processed.

Now, let's write an indicator that uses a *While…Begin* loop to calculate the week-to-date trade volume. Create an indicator named *_VolumeWeek* and type the following EasyLanguage instructions:

```
variable: DaysAgo(0), TotalVolume(0) ;

TotalVolume = Volume ;
DaysAgo = 1 ;

while DayOfWeek(Date) > DayOfWeek( Date[DaysAgo] )
   begin
   TotalVolume = TotalVolume + Volume[DaysAgo] ;
   DaysAgo = DaysAgo + 1 ;
   end ;

Plot1(TotalVolume) ;
```

Example 5-1. Indicator *_VolumeWeek*.

The *TotalVolume* variable holds the total volume for the week and the *DaysAgo* variable is used to reference previous bars. We use the *DayOfWeek* function to get a numeric value for each week day, where Monday is 1 and Friday is 5. As long as the current bar's day is greater than any previous bar, the loop adds the previous bar's volume to the total. For example, on Thursday, the current bar's day value is 4 and the loop adds the volume for Wednesday (3), Tuesday (2), and Monday (1) of the same week. The condition is false when the previous bar's day value is 5 which means that the loop has reached Friday of the prior week.

Now, verify the indicator and apply it to your sample chart in TradeStation using a Histogram format style. The resulting indicator shows increasing volume bars for each day during a week (Figure 5-1).

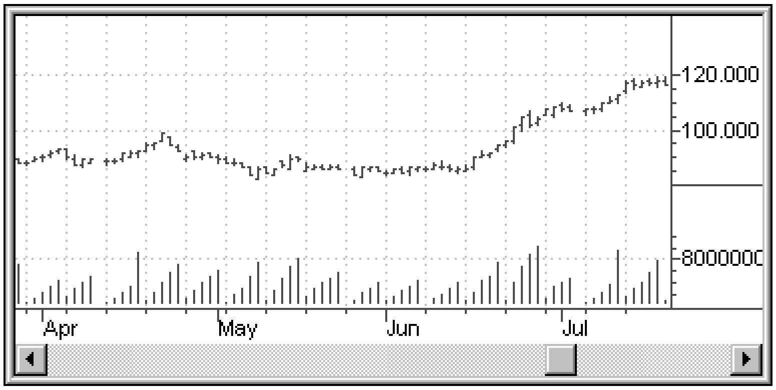


Figure 5-1. Indicator *_VolumeWeek* .

Series Functions

A *series function* is an advanced type of function that refers to itself within its calculations. The capability to reference a previous value makes it quite easy to create a function that maintains a running total. For example, in this simple one-line function:

```
VolTotal = VolTotal[1] + Volume ;
```

the current bar's volume is added to the total that was calculated by the function on the previous bar (*VolTotal*[1] is its value 1 bar ago).

While this might seem complicated at first, it's possible because of the way that EasyLanguage lets you refer to data from previous bars. This type of calculation is also called *recursive* because it refers to itself over and over again. It is best used only by experienced EasyLanguage developers.

More About Variables

Arrays

While a regular variable stores a single value (either numeric or true/false), an array lets you store multiple values under the same name. When you declare an array, you must specify the maximum number of elements (values) that can be stored, and an initial value for all of the elements. Use square brackets to specify the number of array elements and parentheses for the initial value of each element. For example, the following statement:

```
Array: Prices[3](0) ;
```

declares an array named *Prices* that will contain 3 elements, each of which has an initial value of zero. When used in calculations, each array element is referenced by adding the element number after the array as follows:

```
Prices[1]= 100 ;
Prices[2]= 200 ;
Prices[3]= 300 ;
```

...where element 1 is assigned a value of 100, element 2 becomes 200, and element 3 stores 300. Arrays are often used in loops to store related values across a range of bars. In fact, EasyLanguage price data values such as *Close* and *Volume* are actually a type of array, where the element number refers to the "number of bars ago."

Additional Resources

While this book provides you with a general introduction to using EasyLanguage, it is not intended to be a complete reference manual. To help you get the most out of EasyLanguage, a variety of additional resources are available.

TradeStation Help

The TradeStation Help provides documentation on the purpose and use of the EasyLanguage functions and analysis techniques as well as instructions on applying strategies, studies, and indicators to your TradeStation charts. It contains a detailed explanation of each built-in indicator, study, and strategy.

EasyLanguage Support

To assist customers in learning how to use EasyLanguage to accomplish specific goals, the EasyLanguage Support Department offers written answers to questions that are submitted by e-mail, fax, or standard mail. For example, if you are having problems understanding a particular calculation or comparison, the EasyLanguage Support Department can provide you with a plain language explanation of the approach along with sample EasyLanguage statements. You can reach the EasyLanguage Support Department at the e-mail address:

EasyLanguage@TradeStation.com

The EasyLanguage Support Department is not equipped to create custom studies or strategies and does not troubleshoot EasyLanguage procedures written by you or third-party sources.

Exercises and Review

Review

A **Qualifier** is used to specify an alternate data source (**Data**1...**Data**50) for standard price values such as Close, Open, OpenInt, etc. By default, these price values assume **Data**1.

Strings are a data type used to store text.

If...Then...Else statements perform one action when a condition is true and an alternate action when the same condition is false.

For...Begin is a loop that performs a set of actions a specified number of times based on the value of a counter.

A **While...Begin** loop repeatedly performs actions for as long as the controlling condition is true.

Series refers to an advanced type of function that refers to previous values of itself within its calculations.

An **Array** is a special type of variable that allows you to store a series of values under the same name and to use a number (an index) to tell them apart. Arrays are often used with loops to store values based on successive calculations that use a counter as an index number. Because EasyLanguage allocates space for each index number, avoid declaring an array larger than necessary.

Exercises

(Answers are contained in Appendix A)

I. Mark the following either True or False (T or F).

1. A qualifier is used to a change the data source.
2. Values in an array cannot be changed.
3. An *If…Then…Else* statement is a type of loop.
4. A *While…Begin* loop is only executed when a condition is true.
5. An array must have at least 10 elements.
6. The counter variable in a *For…Begin* loop is always incremented by 1.
7. An array index can be a variable.
8. With *If…Then…Else,* an action is taken when a condition is true or false.

II. Identify what type of structure is described using the letters below:

A. For…Begin	C. If…Then
B. While…Begin	D. If…Then…Else

1. You want to count the number of bars that closed higher than they opened over the past 10 bars.
2. An action is performed only when a condition is true.
3. The same action is repeated as long as the close is greater than a bar ago.
4. One action is performed when High < High[1] and another when High > High[1].
5. A moving average is calculated over the last 7 bars.
6. A buy signal is generated after 3 days of an up trend.

CHAPTER 6

ShowMe the Strategy

In this chapter, you'll examine the conditions used in a study and learn how they can be used as the basis for entry and exit strategies. Rather than showing you a single example, this chapter builds on a basic idea and explains alternate methods for expressing conditions and values. The important concept to understand is that there is no one 'right' way of doing things with EasyLanguage, but, rather, it provides you the flexibility to create statements that reflect any level of sophistication, from simple bar comparisions to advanced programming constructs.

The material and examples in this chapter are intended to provide you with a general understanding of EasyLanguage and should not be viewed as an endorsement of any particilar analysis techniques or trading ideas. It is recommended that you complete the exercises and review questions at the end of this chapter to reinforce your learning.

In This Chapter

ShowMe and PaintBar Studies

One of the common characteristics of ShowMe and PaintBar studies is that they mark bars based on a condition. A ShowMe study places the marker above or below the bar on which the condition is true, while a PaintBar study changes the color of a bar that meets a specified criteria. The important thing to note is that, in both types of studies, a bar is marked based on whether a conditional statement is true for that bar. This makes studies a good place to start when looking for conditional ideas that might be useful in trading strategies.

Key Reversal ShowMe - Previous Bar

Let's start with a common bar pattern called a Key Reversal that indicates a change in price movement. First, we'll look at a Key Reveral Up where the Low price of the current bar is down from the previous bar and the Close of the current bar is up from the previous bar.

Example 6-1. ShowMe Study *Key Reversal Up*.

```
if Low < Low[1] and Close > Close[1] then
    Plot1( Low, "KeyRevUp") ;
```

This EasyLanguage statement consists of a two part condition, both parts of which must be true for the plot statement to be executed. Part 1 compares the *Low* of the current bar to the *Low* the previous bar (using the notation *Low[1]* to mean "1 bar ago"). The second part of the statement compares the *Close* of the current bar to the *Close* of the previous bar (again using *Close[1]* to mean "1 bar ago"). The use of the word *and* in the conditional statement indicates that both Part 1 and Part 2 of the statement must be true for the entire condition to be judged true...and only if the entire condition is true will the statement following the *then* clause be executed, in this case *Plot1(Low, "KeyRevUp")*.

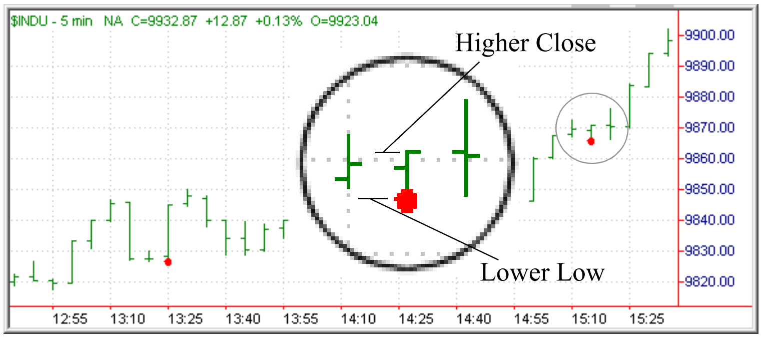


Figure 6-1. Key Reversal Up ShowMe study based on previous bar (1 bar ago).

The chart shown in Figure 6-1 includes a marker on each bar where the EasyLanguage conditions for the ShowMe are true based on the *Low* and *Close* of the previous bar. Note that the position of the ShowMe marker is determined by the first parameter in the Plot1 statement and doesn't have to be just the Low or High. For example, to have a ShowMe marker appear slightly below a bar, simply use any numeric value in the plot statement such as *Plot1(Low-2,"Label")*.

Key Reversal ShowMe - *Lowest* Function

Instead of simply looking at the Low of the previous bar, let's say that you wanted your Key Reversal Up to be based on the lowest Low over the previous 3 bars. To do this, we'll use an EasyLanguage function called *Lowest* that looks for the lowest price over a range of bars in place of the *Low[1]* value used in last example.

Example 6-2. ShowMe Study *Key Reversal Up* using Lowest function

Function name Input parameters Begins looking from previous bar

```
if Low < Lowest(Low,3)[1] and Close > Close[1] then
    Plot1( Low, "KeyRevUp") ;
```

The *Lowest* function includes two parameters, the first of which is the price, *Low* in this case, and the second parameter is the number of bars, *3*. Notice the *[1]* suffix at the end of the function is used to begin looking for the lowest bar starting on the previous bar. Without the *[1]* suffix, the Lowest function would include the current bar in the range which is not what you want to do.

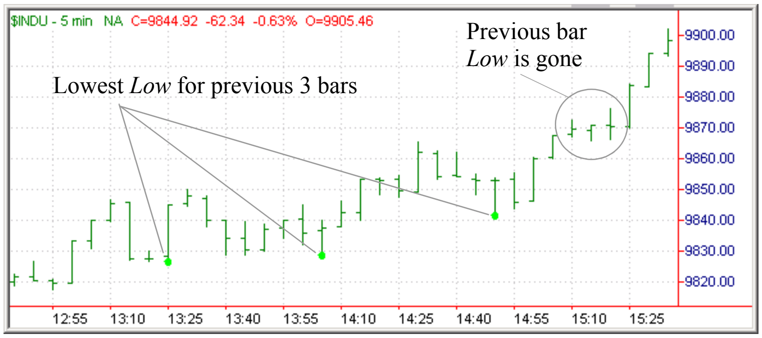


Figure 6-2. Key Reversal Up ShowMe using *Lowest(Low,3)[1] function.*

Looking at the same chart data, notice that their are fewer ShowMe markers than in the previous figure. That's because we're using a slightly different comparison based on the lowest *Low* over the previous 3 bars instead of just looking at the previous bar's Low in the earlier example.

Key Reversal ShowMe - An Input

Now, let's look at how you can use an input value to further improve the flexibility of a ShowMe by allowing you to select an inital comparison value at the time the ShowMe is applied to the chart.

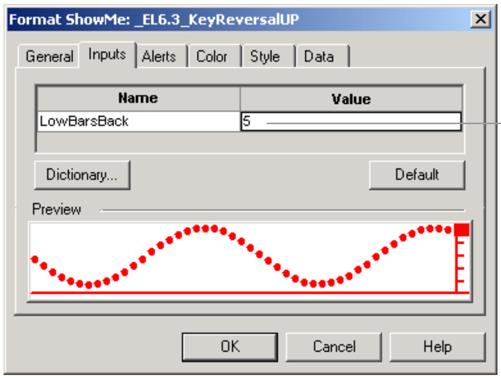
Example 6-3. ShowMe Study *Key Reversal Up* using input

Input declaration and name — Initial value set to 5

```
Input: LowBarsBack( 5 );

if Low < Lowest(Low,LowsBarBack)[1] and Close > Close[1] then
    Plot1( Low, "KeyRevUp") ;
```

The *Lowest* function is still used within the conditional statement, but the second parameter in the function has been changed from a fixed value *3* to an input variable named *LowBarsBack*. This allows you to change the initial value of *LowBarsBack* using the ShowMe properties dialog (Figure 6-3) when you apply it to the chart.



Name	Value
LowBarsBack	5

Initial value can be changed by user

Figure 6-3. Key Reversal Up ShowMe input dialog.

In this example, the bars back range has been set to *5* which further reduces the number of ShowMe markers (Figure 6-4) because it's looking for the lowest *Low* over the previous 5 bars which typically happens less often. The important thing to recognize is the similarity between these first several ShowMe examples...they all test for a Key Reversal Up condition with a lower Low and a higher Close. Where they differ is in the number of bars used for the Low comparison and whether a fixed value or input value is used .

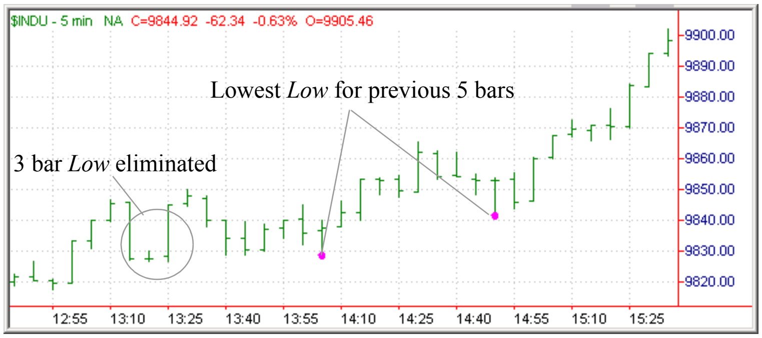


Figure 6-4. Key Reversal Up ShowMe based on Lowest Low for 5 bars..

The Strategy Please

Key Reversal Strategy

A ShowMe study places a marker on each bar based on a condition, so it's a perfect place to begin when creating a long entry strategy based on the same idea. In fact, all you need to do with EasyLanguage is replace the *Plot1* statement from the ShowMe with a *Buy* statement to make the last ShowMe example into an entry strategy.

Example 6-4. *Key Reversal LE* strategy using input

```
Input: LowBarsBack( 5 );

if Low < Lowest(Low,LowsBarBack)[1] and Close > Close[1] then
   Buy("KeyR") next bar at Market ;
```

Buy statemennt replaces the *Plot1*.

When both parts of the condition are true, TradeStation will display a prompt to place a *Buy* order on the next bar at the opening Market price.

It should be pointed out that a *Buy* or *Sell* order in EasyLanguage is actually a 'recommendation' to buy or sell based on the result of one or more conditions in your EasyLanguage statements. The actual 'order' must still be placed by you using either the TradeStation automated entry system or with an external broker.

Since it's fairly common to exit from a position when the entry condition has reversed, let's add a long exit strategy named Key Reversal LX from the TradeStation library. The Key Reversal LX contains a condition that looks for the highest High over the last 5 bars and a lower Close than the previous bar. This is exactly the opposite condition to the entry strategy and TradeStation will generate a prompt to place Sell order (to exit from the long position) when the condition is true.

The Sell order will be placed on the next bar as a stop order using the Low price of the current bar minus 1 point In this case, a stop order is actually a confirming condition that will only be executed if the target price is reached. The stop order only remains active for the designated bar and is considered ’filled’ if the order price occurs (in this case, it becomes fillled if the price drops to the Low-1 point level).

When the entry and exit strategies are applied to the chart (Figure 6-5), an upward arrow appears under bars where an entry position is identified and a downward arrow appears beneath the bar that exits from the position. Notice that the labels underneath both the entry and exit arrows are those that were specified in the EasyLanguage statements for Buy and Sell in the examples above.

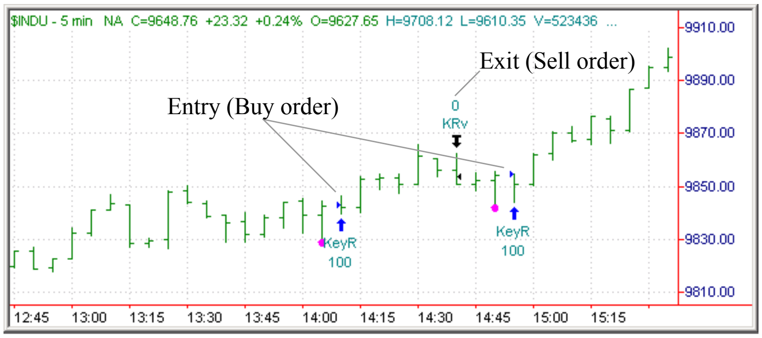


Figure 6-5. Key Reversal Up Strategy..

Summary

As you’ve seen, the logic required to display a ShowMe marker or change a series of bar colors using a PaintBar study is very similar to the logic used in writing strategy conditions. The important difference is that ShowMe and PaintBar studies are used to change the appearance of bars on a chart while Strategies generate Buy or Sell orders that are used to establish or close trading positions. However, the value of using ShowMe and PaintBar studies to visually identify and understand trading ideas cannot be over-stated. The examples in this chapter only scratch the surface, so if you’re interested at looking at more ideas, you should take some time to review the definitions and EasyLanguage scripts for the ShowMe and PaintBar studies contained in the TradeStation library.

Exercises and Review

Review

A **ShowMe** study places a marker above or below any bar that matches the conditions stated in the ShowMe procedure.

A **PaintBar** study changes the color of bars that match a stated condition.

A **Key Reversal** is a common bar pattern that can be used to flag a change in price movement.

Inputs allow you to specify an initial value when an analysis technique or strategy is applied to a chart.

Functions are useful for performing a defined set of calculations that return a value.

Exercises

(Answers are contained in Appendix A)

I. Identify what type of structure is described using the letters below:

A. ShowMe Study	C. Strategy
B. PaintBar Study	D. Function

1. A range of up trending bars are colored blue.
2. The value of the Lowest Low over the past 7 bars is assigned to a variable.
3. A buy signal is generated after 5 days of an up trend.
4. A mark is placed above each bar on a daily chart showing a new weekly high.
5. Yellow bars show increasing momentum and red bars show decreasing momentum.
6. Sell signals are generated after a Key Reversal High.
7. A red marker appears above each Gap Down bar on a chart.
8. A 9 bar moving average calculation is used in an *if* condition.

CHAPTER 7

Crossing Over

This chapter will show you how to create EasyLanguage instructions for simple indicators and how to translate them into strategies. In the process, you will learn more about the use of functions and inputs to increase the flexibility of your indicators and strategies. Also, you will be introduced to the use of multiple exit strategies and learn additoinal EasyLanguage statements that are associated with positions and orders.

The material and examples in this chapter are intended to provide you with a general understanding of EasyLanguage and should not be viewed as an endorsement of any particilar analysis techniques or trading ideas. It is recommended that you complete the exercises and review questions at the end of this chapter to reinforce your learning.

In This Chapter

Indicators and Strategy Elements

An indicator is a type of analysis technique that, generally, plots calculated values on a chart instead of simple marking bars that meet a specified condition. The plotted values might be in the form of a graph line that is overlaid on the bars of a chart or could be a historgram plotted on a sub-graph beneath the bars.

Single Line Moving Average Indicator

A single line moving average is a frequently used analysis tool that calculates the average price of a specified range of bars and plots that average as a wavy line overlaid on top of a set of bars. The EasyLanguage used to plot a single moving average appears below.

Example 7-1. Single Line Moving Average indicator.

```
Plot1( AverageFC( Close,9 ), "MovAvg") ;
```

This EasyLanguage statement plots a single line on a chart (Figure 7-1) based on two parameters, the *Close* price and a range of *9* bars. The EasyLanguage *Plot1* statement is executed for each bar on the chart to obtains the average closing price of the 9 bars before the current bar. The continuous moving average plot line is the result of connecting each bar's average close from one time interval to the next. In addition, the actual moving average line on a chart is colored to help it further stand out from the bars.

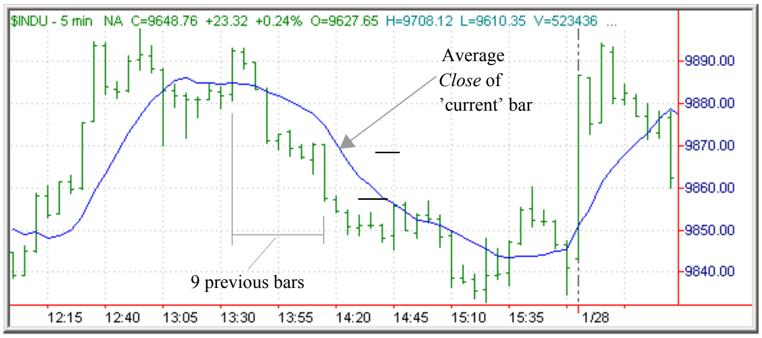


Figure 7-1. Indicator: Single line moving average.

To make this indicator more flexible, you can replace the fixed value parameters in the AverageFC function with input variables that are initialized to the *Close* price and a length of *9* bars.

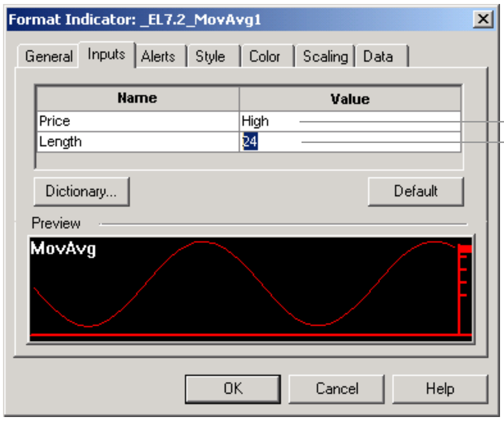
Example 7-2. Single Line Moving Average indicator using inputs.

```
inputs: Price(Close), Length(9) ;

Plot1( AverageFC( Price, Length ), "MovAvg" );
```

$\underbrace{\texttt{AverageFC}}_{\text{Function name}}$ $\underbrace{\texttt{Price, Length}}_{\text{Input parameters}}$

With inputs, you can easily change the values at the time you apply the indicator to a chart...for example, you might want to plot the average of *High*s over a *24* day period as shown in Figure 7-2.


Format Indicator: _EL7.2_MovAvg1

General | Inputs | Alerts | Style | Color | Scaling | Data

Name	Value
Price	High
Length	24

Initial values changed by user

Figure 7-2. Indicator dialog for changing input values when applied to chart

Entry Strategy Based On Single Line Moving Average

The moving average indicator plots a continuous line that is overlaid on the bars...so how could you use a moving average to trigger a buy or sell event in a strategy? All you need to do is add an *If* comparison (Example 7-3) in place of the *Plot1*statement in the indicator! Notice that it compares the result of the *AverageFC* function (using inputs for *Price* and *Length*) with the current bar's *Price* (also based on the type of price specified by the input). You also need to add an action statement following the *then* keyword...in this case, a *Buy* order labeled "MA Entry" is added.

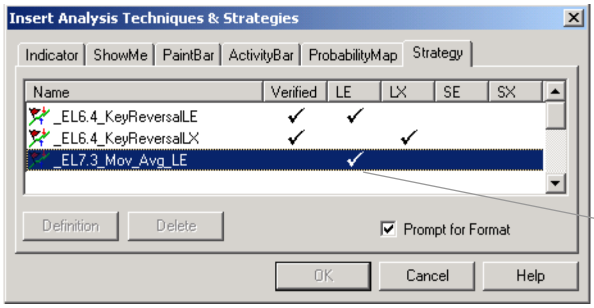
Example 7-3. Strategy based on a Moving Average comparison with the current bar *Price*.

```
inputs: Price(Close), Length(9) ;

if Price > AverageFC( Price, Length ) then
     Buy( "MA Entry" ) next bar at market;
```

If condition replaces the *Plot* statement

When the current bar's price is greater than the calculated average price for the last N bars, a *Buy* order will be generated for the next bar at a market price. Because this strategy includes a Buy statement, it is considered a long entry strategy and will be marked as such in the *Insert Strategies* dialog (Figure 7-3).



Name	Verified	LE	LX	SE	SX
_EL6.4_KeyReversalLE	✓	✓			
_EL6.4_KeyReversalLX	✓		✓		
_EL7.3_Mov_Avg_LE		✓			

Marked as LE (*Long Entry)*

Figure 7-3. Strategy is marked as LE (Long Entry) when it contains a *Buy*

Two Line Moving Average Indicator

Now let's take the idea one step further by using a pair of moving average functions to create an indicator that displays two moving average lines using different range lengths.

Example 7-4. Two Line Moving Average Indicator using inputs and variables.

```
inputs: Price(Close), FastLength(9), SlowLength(18) ;

Value1 = AverageFC(Price, FastLength) ;
Value2 = AverageFC(Price, SlowLength) ;

Plot1( Value1, "FastAvg" ) ;

Plot2( Value2, "SlowAvg" ) ;
```

We'll name the first length input *FastLength* because it is quicker to react to price changes since it is based on average of fewer bars, in this case *9* bars. The second length input is named *SlowLength* and is initially set to *18* bars, which means it will appear to react more slowly to price changes. For this example, we'll assign the result of each *AverageFC* function to variables *Value1* and *Value2* which are then plotted. The reason for using variables will be apparent as we continue on to the strategy example based on the same pair of moving averages.

When this indicator is plotted on a chart (Figure 7-4) it displays two moving average lines that weave their way across the bars. Notice how the fast moving average (solid line) crosses above and below the slow moving average line (dashed line) in response to rising and falling prices. The fast line crosses above the slow line when prices are rising more quickly and crosses below when prices are beginning to decline.

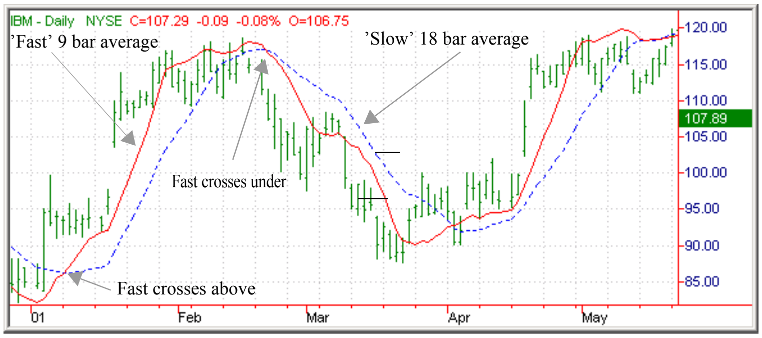


Figure 7-4. Indicator: Two Line Moving Average.

Entry and Exit Strategies Based on Two Line Moving Average

Let's create a strategy from the previous two line Moving Average indicator. All that you need to do is replace the *Plot1* and *Plot2* statements with an *If* statement that tests for a crossover condition. First we'll make an Long Entry Strategy that will Buy when the fast line crosses above the slow line (indicating upward price movement). Then we'll make a matching Long Exit Strategy that Sells when the fast line crosses under the slow line (indicating downward price movement).

Example 7-5. Long Entry Strategy using a two line moving average crossover.

```
inputs: Price(Close), FastLength(9), SlowLength(18) ;

Value1 = AverageFC(Price, FastLength) ;
Value2 = AverageFC(Price, SlowLength) ;

if Value1 crosses above Value2 then
   Buy("MA2 Entry") this bar on close ;
```

For the Long Entry Strategy, we'll use an *if* condition that is true when the price *Value1* (based on the fast 9 bar average of Close) crosses above the price *Value2* (the 18 bar average). Notice that, by using the variables *Value1* and *Value2,* we've made an easy to read condition where the EasyLanguage phrase 'crosses above' reflects the visual appearance of the fast line (*Value1*) crossing above the slow line (*Value2*) on the chart. When the condition is true, a *Buy* order is generated using the *Close* of the current bar as the order price. Remember, EasyLanguage doesn't actually place the order, but simply displays an up arrow mark beneath each matching bar (Figure 7-5) to indicate that it is the start of a long position based on the *Close* of the current bar. The up arrow mark is labled "MA2 Entry" along with the number of shares added to this position. In addition, if the 'current' bar is the last trade on a real-time chart, you'll be shown a alert box that instructs you to place a Buy order for the specified shares using the TradeStation order bar or by calling your broker.

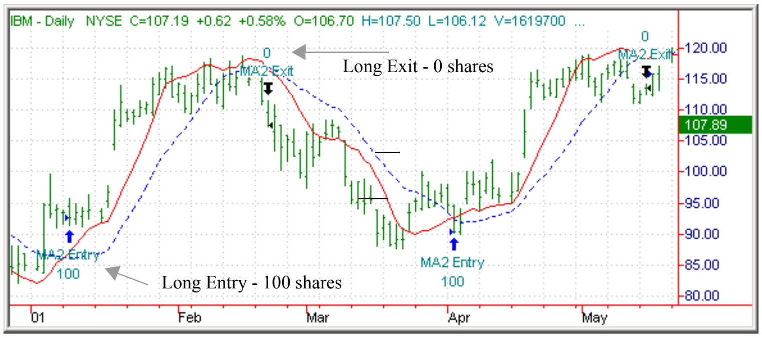


Figure 7-5. Long Entry and Exit Strategies: Two Line Moving Average.

A Long Exit Strategy can be used to exit a long position based on the fast moving average crossing below the slow which indicates downward price movement. All you need to do to change the EasyLanguage is to modify the *if* condition to use the 'crosses below' comparison phrase and to replace the *Buy* order with a *Sell* order (Example 7-6). When the condition matches for a given bar, a downward pointing 'exit' arrow will display above the bar with a label and resulting share position. In this example, the Sell order cancels the previous long position which results in *0* shares held (Figure 7-5) displayed above the "MA2 Exit" label for the sell arrow.

Example 7-6. Long Exit Strategy using a two line moving average crossover.

```
inputs: Price(Close), FastLength(9), SlowLength(18) ;

Value1 = AverageFC(Price, FastLength) ;
Value2 = AverageFC(Price, SlowLength) ;

if Value1 crosses below Value2 then
   Sell("MA2 Exit") this bar on close ;
```

Multiple Exit Strategies

It's quite common to use more than one exit strategy to get out of a position. For example, let's look at adding a stop loss exit to protect against a sudden unforseen change in market conditions. Even though the 'MA2 Exit' is intended to exit the position when the fast average crosses under the long, this could take several bars to develop and might result in a loss that exceeds your risk tolerance. So let's create a basic stop loss exit stategy that will exit from any long position that has dropped at least $250 from it's initial entry price (Example 7-7).

Example 7-7. Stop Loss Long Exit Strategy.

```
inputs: DollarRisk( 250 ) ;
variables: RiskCalc( 0 ), OrderPrice( 0 );

if MarketPosition = 1 then
   RiskCalc = DollarRisk / CurrentContracts ;
   OrderPrice = EntryPrice - RiskCalc ;
   Sell( "$SL" ) next bar at OrderPrice stop ;
end ;
```

The *if* condition *MarketPosition = 1* is used to determine if you're already in a long position...in other words, the statements under the *if* will only be executed if a previous long entry position was generated but not closed. So, when the *if* condition is *true*, two calculations are performed to determine the stop order price...the first calculation determines the risk per share based on the maximum acceptable dollar loss (*DollarRisk*) divided by the number of shares or contracts in the current long position (*CurrentContracts*). Once the risk per share value is calculated (*RiskCalc*), it is substracted from the initial entry price for the position (*EntryPrice*) to produce the new target stop price (*OrderPrice*) that is used in the Sell stop order.

Notice that *DollarRisk* is an input value that can be set by the user when the strategy is applied to a chart while both *RiskCalc* and *OrderPrice* are declared variables used to hold temporary values that are calculated in this strategy only. Two other values, *MarketPosition* and *CurrentContracts,* are system variables that are automatically updated within EasyLanguage on each bar. Finally, notice the use of a *Sell* stop order to set the exit price...if that price occurs within the next bar, the Sell order will be completed...if that price is not met on the next bar, the order is canceled.

Summary

So there you have it...we started with a simple moving average indicator and ended up with several potential entry and exit strategies based on the crossover of a fast and slow moving average. As you can see, EasyLanguage is a very flexible tool for both technical analysis and strategy building.

While there is no limit to the number of exit or entry strategies used together, the important thing to understand is that each entry or exit might affect the overall performance of your total strategy. To make it easy to experiment with different combinations, it's a good idea to use input values that allow you to easy change parameters for entries and exits at the time they're applied to a chart.

Exercises and Review

Review

An **Indicator** is a type of analysis technique that is used to plot calculated values on a chart in the form of a graph line or a sub-graph beneath the bars.

A **Moving Average** indicator plots one or more continuous lines on a chart based on the average of a specified price (such as Close) over a range of bars (Length) before each current bar. This indicator is commonly used to show trend changes.

An **Entry Strategy** is used to generate an order than establishes a market position when a specific entry condition is true.

Exit Strategies are used to generate orders than close a position based on specific exit criteria such as a change in entry conditions, risk tolerance parameters, and money management calculations.

Exercises

(Answers are contained in Appendix A)

I. Mark the following either True or False (T or F).

1. Indicators are used to return a value in a condition or comparison.
2. An entry strategy cannot contain inputs.
3. You can only apply one indicator to a chart at a time.
4. Entry and exit strategies cannot be mixed on the same chart.
5. An exit strategy is only valid if a previous position has been established.
6. A moving average indicator must always be used to indicate a trend change.

CHAPTER 8

Counting On Functions

In this chapter, you'll get a chance to look inside several functions to see how they are used to calculate results and can help make your EasyLanguage procedures easier to use and understand. You will be shown examples of common looping constructs that are useful for handling calculations across a range of bars and data. Also, you'll see how to create functions that return multiple values that can be used to further modularize your procedures and help you understand more advanced coding techniques.

The material and concepts in this chapter include intermediate level EasyLanguage examples that may require additional study. Before continuing, it is a good idea to have reviewed the material in earlier chapters and be sure that you are comfortable with basic EasyLanguage statements and constructs. Also, it's recommended that you complete the exercises and review questions at the end of this chapter to reinforce your learning.

In This Chapter

The Function of Functions

At a basic level, functions are nothing more than self-contained EasyLanguage procedures that, typically, perform some calculations and return a result to the calling procedure. For example, a function might be used to obtain the average of several different prices or could be used to calculate values using a standard 'formula'.

While functions can be used in calculations and comparisons to return a value, you cannot assign a value to a function...the returned function value is read-only.

GetAverage Price Function

The best way to explain a function is to look at an example, so let's start with a function named *GetAverage* (Example 8-1) that uses a looping structure to calculate the average of a specified price over the range of several of bars. Something similiar to this function is used in the moving average indicator and strategy to return a multi-bar price average.

Let's look at the EasyLanguage in more detail. There are really three separate sections in this function...*lines 1-5* include the declarations of inputs and variables, *lines 6-10* contain the main calculations and statements, and *line 11* sets the value that the function will return to the calling procedure.

Example 8-1. Function *GetAverage* calculates the average price over a range of bars.

```
1.      inputs:
2.            Price( numericseries ),
3.            Length( numericsimple ) ;

4.      variables:
5.            Sum( 0 ) ;

6.      Sum = 0 ;

7.      for Value1 = 0 to Length - 1
8.            begin
9.                 Sum = Sum + Price[ Value1 ] ;
10.           end ;

11.     GetAverage = Sum / Length ;
```

Lines 1-3 define the inputs that will be used to accept the pair of values included in parenthesis after the function call name, such as *GetAverage(Close,9)*. Notice that the declared inputs have no initial value as in previous procedures, but rather the *Price* and *Length* inputs are declared based on the type of value that they receive. The first input named *Price* uses the keyword *numericseries* to specify an array of values (such as the Close or Open price). The second input named *Length* is declared as a *numericsimple* value that may be either a single numeric constant or variable. So remember, it's important to pay attention to what kind of procedure you're working with since inputs are declared differently in functions than in other cases.

Lines 4-5 are used to delclare the variable *Sum* which accumulate the multi-bar total of prices for the specified number of bars.

The next set of lines (7-10) is actually one compound EasyLanguage statement, called a *for* loop, that will execute a group of statements a specified number of times. First let's look at line 7...the parameter between the keyword *for* and the equal sign is a 'counter' variable (named *Value*1 in this case)...this variable will automatically be assigned a new value each time the loop is executed. The value between the equal sign and the keyword *to* is the initial value, in this case *0*, that will be assigned to the counter. The last value on the *for* line is the number of times that the loop will be executed, in this case *Length-1* times (where Length is specified as an input when the function is called).

Lines 8 and 10 mark the beginning and end of the EasyLanguage statement(s) that are to be evaluated each time the loop is executed. In this example, the statement on line 9 is a simple calculation that accumulates the *Sum* of a specified *Price[n]* across a range of *n* bars starting with *0* and ending with bar *Length-1*.

Finally, after the loop is done executing for the specified number of steps, line 11 is reached which includes a calculation that divides the *Sum* by the number of bars *(Length*) to determine the average bar price. The result of the calculation is assigned to a return variable that has same name as the function, in this case *GetAverage*, that sends the result back to the calling procedure.

What's important to notice about functions is that their input declarations require you to specify the type of data that they'll accept and that a function returns a value to the calling procedure.

For example, the GetAverage function might be used in the following manner:

Example 8-2. Procedure that uses the function named *GetAverage*

```
variables:
      MyVar ;

MyVar = GetAverage(Close,5) ;
```

where the function *GetAverage* will calculate and return the average of Close prices for a range of 5 bars from the current bar. The value of the function is then assigned to the variable *MyVar*.

Looks Can Be Deceiving

Sometimes it's desireable to want to return more than one value from a single function. So the first question is, how is it possible to do this since a function appears to be able to only return a single named value.

Look at this example of a procedure that uses a function with 4 parameters:

Example 8-3. Procedure that includes a 4 parameter function

```
1.      variables:
2.             MyVar(0),HighVal(0),HighBar(0) ;

3.      MyVar = GetHighest(High,10,HighVal,HighBar) ;
```

At first glance, it looks like this function simply returns its value on Line 3 as *MyVar*...but, in reality, it actually returns 2 additional values to the variables *HighVal* and *HighBar*. The first two parameters in the function, *High* and *10,* are simple inputs that are used to pass values to the function, but that last two parameters, *HighVal* and *HighBar*, are *reference* inputs that are used to return values from the function. Unfortunately, there's no way to tell that this is the case by simply looking at the function from the calling procedure. We need to look deeper into the EasyLanguage statements within the function to understand how this is done.

GetHighest Function

Let's take a closer look at a function named *GetHighest* (Example 8-4) that is designed to find the highest *Price* over a range of *N* bars. However, instead of just returning a single value, it is designed to return both the highest price found and the the relative number of bars back on which that highest price occurred.

Lines 1-5 define the inputs that are used to accept the parameters included in parenthesis after the function call name, such as *GetHighest(Open,15,HighVal,HighBar)*. Remember, with function inputs you must declare the type of value they are to receive rather than specifying an intial value as with non-function inputs. The first input named *Price* uses the keyword *numericseries* to specify that it may accept an array of values (such as the Close or Open price). The second input named *Length* is declared as a *numericsimple* value that may be either a single numeric constant or variable. Now here's where it gets interesting...the third and fourth input values named *oHighVal* and *oHighBar* are declared as *numericref* values which means that their values are actually stored elsewhere and are simply referenced by the function. So what does that mean! If you look back to the *GetHighest* function call in Example 8-3 you'll see that the third and fourth parameters are specified as variables *HighVal* and *HighBar*. Normally this would mean that the value of each variable is passed into the function as a read-only value and used in calculations within the function. However, in the special case of a *numericref* type input, the value of the calling variable is actually 'pointed' to by the function which allows it to save a value to the location of the original variable in the calling procedure. That means that the *numericref* inputs declared on lines 4 & 5, named *oHighVal* and *oHighBar,* can directly read and save values to the variables named *HighVal* and *HighBar* that are specified as the third and fourth parameters in the calling procedure.

It's this special *numericref* input type that makes it possible for a function to return more than one value. To help identify the different role these reference input play in our function we've added an '*o*' (since they are really input/output values) to the beginning of each variable name.

Example 8-4. Function *GetHighest* returns multiple values using pass-by-reference inputs.

```
1.     inputs:
2.        Price( numericseries ),
3.        Length( numericsimple ),
4.        oHighVal( numericref ),
5.        oHighBar( numericref ) ;

6.     variables:
7.        MyVal( 0 ),
8.        MyBar( 0 ) ;

9.     MyVal = Price ;
10.    MyBar = 0 ;

11.    for Value1 = 1 to Length
12.       begin
13.          if (Price[Value1] > MyVal ) then begin
14.             MyVal = Price[Value1] ;
15.             MyBar = Value1 ;
16.          end ;
17.       end ;

18.    oHighVal = MyVal ;
19.    oHighBar = MyBar ;

20.    GetHighest = 1 ; { always returns 1; only outputs used }
```

On lines 6-8, two local variables named *MyVal* and *MyBar* are declared with initial values of '0' in both cases. These will be used to store temporary values used in the functions calculations.

The main portion of the function, lines 11-16 contain a *for* loop which operates in a similar manner to the loop described in Example 8-1. In this case, the loop will execute a group of statements a specified number of times and updates the counter *Value1* with a starting value of '1' and an ending value of *Length*. This is slightly different than in the previous function example because this function is designed to look at a range of bars that starts with the bar before the current bar instead of including the current bar in the range.

The *begin* and *end* on lines 12 and 16 mark the start and end of the EasyLanguage statements that will be executed with each loop iteration. Lines 13-16 includes an *if* condition that is true when the *Price* of bar *Value1* is greater than the last value of *MyVal*. Since the inital value of *MyVal* is '0', this condition should always be true for the *Price* of bar 1. When true, the calculations on lines 14 & 15 are performed which saves the new 'highest' price value of the bar being tested as *MyVal* along with the actual bar number as *MyBar*. As the loop progresses, the *Price* of each bar in the range is tested against the value of *MyVal*. When the loop is finished, the variable *MyVal* will contain the highest price found and *MyBar* will hold the bar position of that price.

Finally, lines 18-20 are used to set the return values. The highest price and bar position are going to be returned using the special *numericref* inputs. However, to properly verify, a function must also include a return variable that has same name as the function, in this case *GetHighest*. In this case, we'll simply always return a value of '1' since the primary calculated values have already been returned using *oHighVal* and *oHighBar*.

Summary

Functions can be thought of as "intelligent" values that perform calculations based on specified input parameters. For example, the *GetAverage(Close,5)* function, explained in Example 8-1, returns the average of the *Close* prices across a range of *5* bars, including the current bar. However, by changing the parameters to *GetAverage(High,12)*, the same function can be used to calculate and return more than one value, in this case the average of *High* prices across a range of *12* bars. A good rule to follow is that if you find yourself performing the same type of calculations over and over again in your procedures, you might want to develop a function that performs those same calculations. It not only helps simplfy the code in your main EasyLanguage procedures, but it helps ensure that the calculations are consistent and 'tested' because, once developed, the function will always perform them in a uniform manner. If, later, you decide to change or refeine these calculations, all you need to do is edit the function and all of the analysis techniques and strategies that use the function will reflect the change as well.

Exercises and Review

Review

A **Function** is a EasyLanguage script or procedure that performs calculations on specified input parameters and return results to the calling procedure.

The **for** loop allows the same EasyLanguage statements to be executed a specified number of times using a counter variable.

A **Return** statement is a required element at the end of a function that is used to pass back a value and which has the same name as the function.

Exercises

(Answers are contained in Appendix A)

I. Mark the following either True or False (T or F).

1. Loops may only be used in functions.
2. You may assign a value to a function just like any other variable.
3. Function inputs are exactly the same as indicator or strategy inputs.
4. Inputs may also be used to return values from a function
5. A function can be applied to a chart along with indicators and strategies..
6. Loop statements must always include *begin* and *end* keywords.
7. A *for* loop is only executed if the counter condition is true.

CHAPTER 9

Filtering Adds Flexiblity

In this chapter, you'll examine the use of filter conditions in a simple strategy to make the strategy more flexible and to be able to create more complex entry and exit rules.

The concepts and examples used in this chapter build on a basic knowledge of entry and exit strategies that are part of the strategy building process that may require additional study. Therefore, before continuing, it is a good idea to have reviewed the material in earlier chapters and be sure that you are comfortable with basic EasyLanguage statements and constructs. Also, it's recommended that you complete the exercises and review questions at the end of this chapter to reinforce your learning.

In This Chapter

Understanding Strategy Elements

You learned about entry and exit strategy elements in several of the earlier chapters in this book. The important thing to recognize is that an entry strategy is used to determine when and how to enter, or establish, a market position (long or short) while an exit strategy either closes or reverses as established position. While many strategies utilize just a single entry strategy element, it’s quite common to have several different exit strategy elements. This is true because you may want to create exit conditions that accomodate money management and risk avoidance calculations in addition to a simple change in the original entry condition.

Now we’re going to take this one step further by looking a strategy element that use several EasyLanguage conditions to determine if a buy or sell order should be generated.

Momentum Long Entry Strategy

Example 9-1 is a fairly simple entry strategy that places a Buy order when the change in momentum rises from the previous bar. Momentum is considered to be a difference in price between two bars that are *Length* bars apart. An increase in price is considered positive momentum while a decrease is price is negative momentum. However, in this example we’re actually looking to see if momentum is positive and has become MORE positive since the previous bar.

Example 9-1. Strategy - Momentum long entry.

```
1.      inputs:
2.            Price( Close ),
3.            Length( 10 ) ;

4.      variables:
5.            Mom( 0 ) ;

6.      Mom = Price - Price[Length] ;

7.      if Mom > 0 and Mom >= Mom[1] and MarketPosition <> 1 then
8.            Buy( "Mom" ) next bar at High + 1 point stop;
```

The default input values are declared in Lines 1-3 with the *Price* set to Close and a momentum *Length* of 10 bars. Remember, because these values are inputs, they can be changed when this strategy is applied to your chart. Lines 4-5 are used to declare a variable that will hold the calculated value for momentum which is defined as the difference between two prices that are *Length* bars apart. Line 7 contains an *if* condition that has three separate comparisons that must all be true for the entire condition to be true. The first comparison tests whether the value for *Mom* is greater than zero since we’re looking for a position change in momentum. The second comparison, *Mom >= Mom[1]*, is the key to this strategy...it’s testing to see if the momentum from *Length* bars ago to the current bar (Mom) is greater than, or equal to, the momentum from one bar ago for the same *Length* number of bars (*Mom[1]*). EasyLanguage is smart enough to automatically calculate values for *Mom* and *Mom[1]* based on the assignment statement in Line 6, even though you’ve only specified a single calculation. The final comparison, MarketPosition <> 1, will only be true if we are not already in a long position. Again, no sense in continuing if we are already long!

So, if all three comparisons in the *if* statement are true, the EasyLanguage statement(s) following the keyword *then* is executed, in this case Line 8. This statement actually generates a *Buy* stop order at the specified price for the next bar. Notice that it sets a stop at one point above the *High* of the current bar which will only be executed if the price continues to rise about the previous bar's high price...thereby confirming upward price movement. This is a classic setup and trigger arrangement, where the *if* condition provides the setup (i.e. are conditions right for this type of trade) while the *Buy stop* order establishes the trigger that will be executed only if the related price move is confirmed.

Obviously, this is a long entry strategy since the resulting *Buy* order places us in a long position. To change this to a short entry strategy, all you'd need to do is change the *if* condition to test for a negative momentum value on the *Low* that becomes more negative from the previous bar to the current bar. Of course, a short entry strategy would place a *Sell* order based on the *Low* price.

Filtering A Strategy

So, in Example 9-1 you can see how a momentum entry strategy operates. Clearly, you could make the test condition more sophisticated by looking at the change in momentum over more than the previous bar (for example, *Mom>=Mom[1] and Mom[1]>=Mom[2]*)...but that's really just an extension of the basic strategy rule of testing for a increase in positive momentum. However, let's say that you wanted to refine this strategy by also checking for trade volume above a specified threshold. This can be done quite easily by adding a filter condition that is evaluated after the primary condition is true but before an order is generated.

Example 9-2 is basically the same as the previous example with the addition of two more lines. First, a new input named *Target* has been added at Line 4 with an initial value of *1000*. This represents the volume threshold that will be used for your filter. Be aware that you need to consider how this volume threshold will be affected by the trading volume of the symbol you're trading and the bar increment on your chart. For example, the trade volume of a heavily trading symbol on a 1 minute bar might be tens of thousands of shares while an small volume symbol might have only hundreds of shares traded on a 5 minute bar.

The actual filter condition has been added as Line 9 as a second *if* statement that tests whether current bar *Volume* is greater then the *Target* input value. So, the only way a *Buy* order will be generated is for the a positive change in momentum to exist at the same time the current bar *Volume* is at or above a specified threshold. This second *if* statement is sometimes called a "nested" statement since it is only evaluated when the previous *if* is true. You could easily added more nested *if* statements beneath the *Volume* test to further restrict the generation of a Buy order based on other conditions.

Example 9-2. Strategy - Momentum long entry with Volume filter.

```
1.      inputs:
2.            Price( Close ),
3.            Length( 10 ),
4.            Target( 1000 ) ;

5.      variables:
6.            Mom( 0 ) ;

7.      Mom = Price - Price[Length] ;

8.      if Mom > 0 and Mom >= Mom[1] and MarketPosition <> 1 then
9.            if Volume >= Target then
10.                Buy( "Mom" ) next bar at High + 1 point stop;
```

Time Filter

So what would another filter condition look like? In Example 9-3, we've added a time of day filter beneath the Volume filter to allow the Momentum long entry strategy to place trades only during the middle of the day. The pair of new conditions have been added at line 10 that will be true if the time of day is greater than 10:30am but less than 3:00pm. Even when the primary Momentum condition and Volume conditions are true, any potential trades occuring before or after the target times will be ignored. Note that all times are specified using a 24-hour format. Also, notice the use of the optional parenthesis around the pair of times that help you recognize that these are really associated conditions. In other words, the Volume condition is our first filter and the two time conditions in parenthesis are the second filter.

Example 9-3. Strategy - Momentum long entry with Volume filter and Time filter.

```
1.      inputs:
2.            Price( Close ),
3.            Length( 10 ),
4.            Target( 1000 ) ;

5.      variables:
6.            Mom( 0 ) ;

7.      Mom = Price - Price[Length] ;

8.      if Mom > 0 and Mom >= Mom[1] and MarketPosition <> 1 then
9.        if Volume >= Target then
10.         if (Time > 1030 AND Time < 1500) then
11.            Buy( "Mom" ) next bar at High + 1 point stop;
```

As you have seen, filter conditions are basically additional trading rules that must be true for an order to be generated. In other words, the primary rule, represented by the first *if* condition, must be true AND all of the filter conditions, represented by nested conditions, must be true for an order to be generated.

So how is this different than simply creating a new entry strategy for each rule and adding it to your chart? The important difference is that entry strategies are independent of one another and any entry strategy can generate an order where filters are co-dependant and both the primary entry rule and ALL filter rules must be true for an order to be generated. To put it in programming terms, multiple entry strategies are a series of OR conditions where a strategy with filters represents a series of AND conditions.

Summary

An entry or exit strategy is based on a primary rule that can also be modified by additional rules called filters. Each filter rule, in effect, represents one more condition that must be true for the strategies buy or sell action to be executed. For example, you might use a filter to limit an entry condition based on trade volume or on the value of an overbought/oversold indicator. The important thing to pay attention to is that you don’t make filter conditions so restrictive that the primary entry or exit rule condition never gets applied. This is especially true if you intend to use exit strategies for risk avoidance or money management...in other words, a low volume price drop is still a loss!

Exercises and Review

Review

An **Entry** strategy contains one more more rules that are used to establish a market position.

An **Exit** strategy is used to close out or reverse an existing position and consists of one or more rules.

A **Filter** condition is typically a secondary rule in a strategy that is evaluated only when the primary Entry or Exit rule is true.

Exercises

(Answers are contained in Appendix A)

I. Mark the following either True or False (T or F).

1. Entry strategies are only used to enter a long position.
2. You must use a filter if you want to evaluate more than one condition in a strategy.
3. The following EasyLanguage expression can be used to determine if a long position had been previously established:

```
MarketPosition <> 1
```

4. You can have many exit strategies with a single entry strategy.
5. Filter conditions must always make positive comparisions.

APPENDIX A

Answers to Exercises

CHAPTER 1 - Answers

I. Match each numbered word with its correct definition. Write the matching letter next to the word's number.

1. – E	**2. – F**
3. – G	**4. – I**
5. – A	**6. – H**
7. – D	**8. – B**
9. - C	

II. Indicate which of the following items are true or false.

1. False – An EasyLanguage procedure always evaluates every bar on the chart.
2. False – Not all reserved words are complete statements.
3. True – Bars on a chart start with the oldest bar at the left of the chart and continue evaluating bars until the newest bar at the right is reached.
4. False – Skip words are ignored by EasyLanguage and do not prevent other Reserved words in an instruction from being processed.

CHAPTER 2 - Answers

I. Mark the following either True or False (T or F).

1. False – Many signals are as simple as an *If…Then Buy* statement.
2. True – Both return values, but only variables can be assigned a value.
3. False – In EasyLanguage, numeric calculation works right to left. This example compares two values in evaluating a conditional (true/false) expression.
4. False – *If…Then* is widely used in all types of analysis techniques and functions.
5. False – The execution method "this bar on close" places orders for the current bar. All other orders (including *or Higher* and *or Lower*) are placed on the next bar.

6. False – A user defined variable name cannot be declared more than once in a procedure. However, once declared, a variable can be used in numerous calculations and assignments.
7. False – Only variables can be assigned a new value after they have been declared.

II. Identify each statement's type using the letters below:

A. Conditional statement

B. Declaration statement

C. Assignment statement

1. C (Assignment).
   This statement assigns a condition (*High>High[1]*) to a true/false variable.
2. A (Conditional).
   The order is placed only if the condition is true
3. B (Declaration).
   This statement declares two numeric variables and sets their initial value to zero.
4. C (Assignment).
   The average of the two closes is stored as Value10.
5. A (Conditional).
   This is a conditional statement that performs two variable assignments when the condition is true. Note the use of the block words *Begin…End* to perform more than one action.
6. B (Declaration).
   This statement declares two input values.
7. A (Conditional).
   Even though the action of this *If…Then* statement is to plot a value, it is still a conditional statement since the plot will only occur if the condition is true.
8. C (Assignment).
   The value returned from the *Average* function is stored in the user variable *SlowAvg* which would have to be previously declared.

III. Write EasyLanguage statements for the following.

1. If today's high is greater then yesterday's close, buy 100 shares of GM at tomorrow's open.
   ```
   If High>Close[1] Then Buy 100 Shares Next Bar at Market;
   ```

2. Buy as soon as the next bar's price is greater than today's high.

```
Buy Next Bar at High + 1 Point or Higher;
```

Note: It's clearer to use the words *or High*er when writing a Buy Stop order such as this.

3. When the current bar closes up from the previous day's high, buy 25 shares at a price of $45 or higher.

```
If Close > High[1] Then
  Buy 25 Shares Next Bar at 45 or Higher;
```

4. When IBM's close is higher than yesterday's by 2 percent, you want to sell another 100 shares.

```
If Close > Close[1]*1.02 Then
  SellShort 100 Shares Next Bar at Market;
```

Note: You may have found that the text description was misleading since it didn't specify when to sell or at what price. It would have been better to add "at the market price of the next bar" to make it clearer. Be careful when preparing your own descriptions that you are as clear and complete as possible.

5. If you are in a long position and today's high is lower than yesterday's close, then you want to exit your position.

```
If MarketPosition=1 AND High<Close[1] Then
  Sell This Bar at Close;
```

Note: This text description was also incomplete since it didn't specify when to exit or at what price. Be careful when preparing your own descriptions that you are as clear and complete as possible.

CHAPTER 3 - Answers

I. Mark the following either True or False (T or F).

1. False - Set-up and Entry is a commonly used technique to determine when and how to enter a position.

2. True - A Buy statement establishes a long position if not in any position, or, if in a short position, closes out the short position and creates a long position.

3. False - In EasyLanguage, a SellShort statement establishes a short position if no other position exists, or, if in a long position, close out the long position and creates a short position.

4. True - This is the default action of EasyLanguage. If you add the word *Total* to the end of the phrase, it <u>only</u> closes out 2 contracts from the first position(s).

5. False -. A multi-data strategy looks at data from more than one <u>data stream</u> at a time on <u>the same</u> chart.

6. False -. The only difference between a regular strategy and a multi-data strategy is the use of additional data streams.

7. False - A trailing stop is used to exit based on a price.

8. False - A strategy does not have to place an order, but it would be unusual not to do so. It's a good idea to consider using an entry with multiple exits as part of your trading strategy.

9. True - An input in a function includes the data type (numeric, true/false) for the value. For analysis techniques (including studies), the default input includes the initial value (either a number or a true/false condition).

II. Identify each order type using the letters below:

A. Enter Long Position	C. Close Out Long Position
B. Enter Short Position	D. Close Out Short Position

1. D (Close Out Short Position).
   This statement closes out a short position at the close of the current bar.

2. A (Enter Long Position)
   This buy order establishes a long position if the condition is true. It also closes out any short position that might have been previously open.

3. B (Enter Short Position)
   If the condition is true, a short position is created and any open long position is exited.

4. A (Enter Long Position)
   Same as 2.

5. C (Close Out Long Position)
   This statement actually places an initial stop order that only closes out a long position if one is also established for the next bar with another buy statement.

6. A (Enter Long Position)
   Establishes a long position if the condition is true and the price is 100 or lower.

7. A and C (Enter Long and Close Out Long)
   Establishes a long position and places a sell stop order at a price of 90% of the Close if the condition is true.

8. A (Enter Long Position)
   Creates a long position.

CHAPTER 4 - Answers

I. Mark the following either True or False (T or F).

1. False – Indicators and studies are both considered analysis techniques.
2. False – A PaintBar study typically changes a bar's color.
3. False – While most indicators and studies include a plot, they are not required to.
4. False – An alert occurs only for the last bar on the chart.
5. True – One plot is needed to set the start price and a second plot to set an end price.
6. False – The default style and scaling for an analysis technique can be set at the time you create it.

CHAPTER 5 - Answers

I. Mark the following either True or False (T or F).

1. True – However, Data1 is assumed if no qualifier is used.
2. False – The value of each array element can be stored and changed just like any simple variable.
3. False – It is just an expanded form of *If…Then*.
4. True – The loop is executed only when the test condition is true.
5. False – An array can have any number of elements.
6. False – The counter can increase in steps of 1 or decrease by –1.
7. True - An array index can be either a fixed number or a variable.
8. True – The first set of statements (following the *If*) are processed when the condition is true, and the second set of statements (following the *Else*) are processed when the condition is false.

II. Identify what type of structure is described using the letters below:

A. For…Begin	C. If…Then
B. While…Begin	D. If…Then…Else

1. A (For…Begin). The following EasyLanguage loop starts with the current bar and looks at the previous nine bars by using BC as the bars ago offset (values 0 to 9). The variable **UpCount** is incremented every time a bar's Close is greater than its open.

```
UpCount = 0 ;
For BC = 0 to 9 Begin
    If Close[BC] > Open[BC] Then UpCount = UpCount + 1;
End ;
```

2. C (If…Then). In the following example, the action is performed whenever the current bar's high is less than the previous bar's high.

```
If High < High[1] Then ACTION ;
```

3. B (While…Begin). This is best handled with a While loop that uses a test condition based on the close. Here is a general example:

```
While Close > Close[1] Begin
    ACTION ;
End ;
```

4. D (If…Then…Else). Although this might look like a situation where you would use two regular *If* statements (one for each condition), this is a case where the stated conditions are actually the reverse of one another. Therefore, it's better to use an If...Then...Else statement where the first action is taken when the condition is true and the second is taken when the condition is false. See the following:

```
If High < High[1] Then
    Action1 ;
Else
    Action2 ;
```

   Remember, you can also use *Begin…End* around a set of statements if your action requires multiple statements.

5. A (For…Begin). Normally when you need to perform an action over a fixed number of bars, the For…Begin loop will do.

```
Variable: Count(0), CloseTotal(0), CloseAvg(0) ;
```

```
For Count = 1 to 7 Begin
   CloseTotal = CloseTotal + Close[Count] ;
End ;

CloseAvg = CloseTotal / 7 ;
```

However, instead of writing your own loop, you could just use the built-in *Average* function to get the same result:

```
CloseAvg = Average(Close,7) ;
```

6. C (If…Then). Using the *_CloseUps* function you created in a previous chapter, you could generate a buy order with a simple *If* statement:

```
If _CloseUps(3) Then Buy This Bar on Close;
```

CHAPTER 6 - Answers

I. Identify what type of structure is described using the letters below:

A. ShowMe Study	C. Strategy
B. PaintBar Study	D. Function

1. B (PaintBar Study). Changing the color of one or more bars is usually done by this type of analysis technique.
2. D (Function). The value of a function is calcuated and returned to the calling EasyLanguage script where it may be assigned to a variable, as in this example.
3. C (Strategy). The EasyLanguage structure that can generates an order is a Strategy element. In this case, an entry strategies was used to establish a position with a Buy order..
4. A (ShowMe). Although other types of analysis techniques might be used to place a mark above or below a bar, a ShowMe structure is the answer here.
5. B (PaintBar). Again, changing the color of bars is a feature of the PaintBar study.
6. C (Strategy). Only strategies can generate Buy and Sell orders.
7. A (ShowMe). Another example of a marker placed by a ShowMe.
8. D (Function). The value of a function can be used in any calculation or comparisoin. In this case, a Moving Average function was part of an *if* condition.

CHAPTER 7 - Answers

I. Mark the following either True or False (T or F).

1. False - A Function returns a value and can be used in a condition or comparison.
2. False - Inputs are often used in entry and exit strategies to set initial parameters.
3. False - TradeStation lets you apply multiple indicators and strategies to a chart.
4. False - In fact, you need both entry and exit strategies to open and close positoins.
5. True - An exit strategy closes a previously established position.
6. False - There are many types of indicators that can be used to indicate a trend change from key reversals to more sophisticated averages and indexes.

CHAPTER 8 - Answers

I. Mark the following either True or False (T or F).

1. False - A loop is a general language construct that can be used in any EasyLanguage script to execute specific statement a number of times.
2. False - A function is a Read-Only value and you cannot be assigned a value in the same manner as a variable.
3. False - The input statement in a function requires that you specify the type of value that is to be received but does not let you set its initial value.
4. True - While it may sound counter-intuitive, you can use a *by reference* input type (also called an Output) to change a value in the calling EasyLanguage script.
5. False - Functions are used in calculations and comparisons and are not directly applied to a chart. Indicators, studies, and strategies are applied to charts.
6. False - The use of *begin* and *end* keywords is only required if multiple EasyLanguage statement are to be executed by a loop or comparison.
7. False - A *for* loop executes a specific number of times as determined by the maximum value of the counter.

CHAPTER 9 - Answers

I. Mark the following either True or False (T or F).

1. False - An entry strategy can be used to establish either a long or short position.
2. False - Entry and exit strategy comparisons often include have complex rules that consist of more than one condition. Filters are simply used to add other conditions based on the truth of the primary entry or exit rule.
3. False - This is a bit of a trick question. The expression MarketPosition <> 1 is true when a long position doesn't exist, but can't really tell you if a long position had been previously opened and closed because this condition would also be true if a long position was never established.
4. True - It's very common to use multiple exit strategies with a single entry strategy. For example, you might have one exit that closes a position when the entry condition is no longer true along with several different risk and money mangement exits.
5. False - A filter condition, just like any other comparison, can make either positive or negative comparisons. However, a filter will only execute the associate EasyLanguage statements when its condition is True.

APPENDIX B

User Functions

A	Note
AB_AddCellRange	Adds multiple cells within a price range to an ActivityBar.
AB_AverageCells	Average number of ActivityBar cells per row.
AB_AveragePrice	Average price of ActivityBar cells on a particular side or over the entire bar.
AB_CellCount	The number of cells on one or both sides of an ActivityBar.
AB_Median	Median value for the ActivityBar cells of the current bar.
AB_Mode	The number of cells in the mode row of the ActivityBar.
AB_NextColor	Changes the color of ActivityBar cells for specified minute-based intervals.
AB_NextLabel	Changes letter in an ActivityBar cell based on minute-based intervals.
AB_RowHeightCalc	The row height to be used for ActivityBar cells.
AB_StdDev	Standard deviation value of the ActivityBars.
AbsoluteBreadth	Calculates market momentum based on the advancing and declining issues.
AccumDist	Examines total daily volume to find the start of a Bull or Bear market move.
AccumSwingIndex	Keeps a running total of the SwingIndex values for a current bar.
AdaptiveMovAvg	Calculates a variable speed exponential moving average.
AdvanceDeclineDiff	The current ratio between advancing issues and declining issues.
AdvanceDeclineRatio	The cumulative difference between advancing issues and declining issues.
ADX	Wilder’s Average Directional Index measures the trending quality of the market.
ADXCustom	Same as ADX, except it lets you specify what value to use for the DMI.
ADXR	Wilder’s ADX Rating of a symbol according to its strength of movement.
ADXRCustom	Same as ADXR, except you specify values for DM Plus and DM Minus.
ArmsIndex	The ratio between adv/declining issues and the adv/declining Volume.
Average	Gets the simple average of the specified data series for the last *N* bars.
AverageArray	Gets the simple average of the specified array.
AverageFC	Uses a fast calculation method to get same value as the Average function.
AvgDeviation	The average of the absolute deviation of data points from their mean.
AvgDeviationArray	The average of the absolute deviation of data points from their mean in an array.
AvgPrice	The average price of a bar calculated by adding OHLC and dividing by four.
AvgTrueRange	An average of the TrueRange values over a period of time.
B	
BarAnnualization	Annualization based on the data compression of a bar.
BarNumber	Provides a bar’s number relative to the start of the chart.
BearishDivergence	Looks for highs in prices not accompanied by highs in an oscillator.
BollingerBand	Calculates the Bollinger Band offset from a specified price or moving average.

BullishDivergence	Looks for lows in prices not accompanied by lows in an oscillator
C	
CalcTime	Adds and subtracts minutes from the time and returns new time in HHMM format.
CCI	Measures the deviation from normal cycles to indicate major trend changes.
ChaikinOsc	Variation of the AccumDist function that measures the direction of a trend.
CloseD	Closing price of *N* days ago from the current bar in an intraday chart.
CloseM	Closing price of *N* months ago from the current bar.
CloseW	Closing price of *N* weeks ago from the current bar.
CloseY	Closing price of *N* years ago from the current bar.
CoefficientR	Calculates the R Coefficient for the past *N* bars.
CoefficientRArray	Calculates the R Coefficient for the past *N* bars for an array.
Combination	Determines the number of combinations for a given number of items.
Correlation	Calculates the correlation coefficient between two data sets.
CorrelationArray	Calculates the correlation coefficient between two array.
CountIf	Counts the number of occurrences of a specified criteria over the last *N* bars.
Covar	Determines the strength of the relationship between two data sets.
CovarArray	Determines the strength of the relationship between two arrays.
CSI	Identifies markets that will likely provide greater returns on dollars invested.
CSIClassic	Classic formula for CSI to determine the likelihood of greater returns.
Cum	Cumulative total of a data series, up to and including the current bar.
D	
DailyLosers	The number of losing positions that were taken throughout the date specified.
DailyWinners	The number of winning trades that were taken throughout the date specified .
DaysToExpiration	The number of days left between today and a stock option's expiration date.
Detrend	Calculates the detrended value of a price over a period of bars.
DevSqrd	The sum of squares of deviations of data points from their average.
DevSqrdArray	The sum of squares of deviations in an array from their average.
DMI	Identifies the amount of directional movement or trend strength quality.
DMICustom	Same as DMI, except you specify the price the function uses.
DMIMinus	Commonly used with DMIPlus to identify an uptrend or a downtrend.
DMIMinusCustom	Same as DMIMinus, except you specify the price the function uses.
DMIPlus	Commonly used with DMIMinus to identify an uptrend or a downtrend.
DMIPlusCustom	Same as DMIPlus, except you specify the price the function uses.
E	
EaseOfMovement	Gauges the magnitude of price and volume movement
ELDate	Returns a date in EasyLanguage format (YYYMMDD)
ELDate_Consol	Returns a date in EasyLanguage format (YYYMMDD)
ELDateToString	Returns a date string from a specified Julian date.

EntriesToday	The number of entries that have been taken on the date specified
ExitsToday	The number of exits that have been taken on the date specified
ExtremePrice	The ratio of the extreme values (high/low) for a length of bars.
Extremes	Provides the value and number of bars ago the most extreme prices occurred.
ExtremesArray	Provides the value and element number of the most extreme values in an array.
ExtremesFC	Uses a fast calculation to get the same number as Extremes.
F	
Factorial	Calculates the factorial of a number.
FastD	Fast percentD value used in Stochastic calculations.
FastDCustom	Fast percentD value based on custom prices used in Stochastic calculations.
FastDCustomOrig	Fast percentD value based on custom prices used in Stochastic calculations.
FastHighestBar	Part of the HighestBar function.
FastK	Fast percentK line used in Stochastic calculations with custom prices.
FastKCustom	Fast percentK line based on custom prices used in Stochastic calculations.
FastKCustomOrig	Fast percentK line based on custom prices used in Stochastic calculations.
FastLowestBar	Part of the LowestBar function.
FindBar	Searches back in time for the first bar matching the date and time specified.
Fisher	Calculates the Fisher transformation.
FisherINV	Calculates the inverse of the Fisher transformation.
G	
H	
HarmonicMean	Calculates the harmonic mean of a data set.
HarmonicMeanArray	Calculates the harmonic mean of an array.
HighD	The high price of *N* days ago from the current bar.
Highest	Finds the highest PRICE value over a period of time.
HighestArray	Finds the highest PRICE value over a period of time in an array.
HighestBar	Finds the number of bars ago when the highest PRICE occurred.
HighestFC	Finds the highest PRICE value over a period of time using fast method.
HighM	The high price of *N* months ago from the current bar.
HighW	The high price of *N* weeks ago from the current bar.
HighY	The high price of *N* years ago from the current bar.
HPI	The money flow in and out of the market or commodity to which it is applied.
I	
IFF	Returns one value if a condition is true and another value if false.
IFFLogic	Returns one logical value if a condition is true and another value if false.

J	
K	
KeltnerChannel	Calculates the Keltner Channel value for a specified bar.
Kurtosis	Calculates the Kurtosis of a data set.
KurtosisArray	Calculates the Kurtosis of an array.
KurtosisOpt	Calculates the Kurtosis for which all the data points are not available.
L	
LastBarOnChart	True if the current bar is the last bar on the chart.
LastCalcDate	Date of the last completed bar, in YYMMDD format.
LastCalcTime	Time of completion (close) of the last bar, in 24-hour military format.
LastDayOfMonth	Last calendar day of the specified month.
LastHour	True if the current time is within the last hour of the first trading session.
Leader	True if mid-point is greater than a previous high or less than a previous low.
LinearReg	Calculates the slope, angle and regression value of the current regression line.
LinearRegAngle	Calculates the angle of the current regression line.
LinearRegAngleFC	Calculates the angle of the current regression line using the fast method.
LinearRegFC	Uses a fast calculation method to derive the same values as LinearReg.
LinearRegSlope	Calculates the slope of the current regression line.
LinearRegSlopeFC	Calculates the slope of the current regression line using the fast method.
LinearRegValue	Calculates the regression value of the current regression line.
LinearRegValueFC	Calculates the regression value of the current reg line using the fast method.
LinRegArray	Calculates the slope, angle and regression value based on an array.
LinRegForecastArray	Calculates the predicted y-value for a given x-value based an two arrays.
LinRegInterceptArray	Determines the point at which a line will intersect the y-axis based on two arrays.
LinRegSlopeArray	Calculates the slope of the linear regression line using two arrays.
LowD	Low price of *N* days ago from the current bar.
Lowest	Finds the lowest PRICE value over a period of time.
LowestArray	Finds the lowest PRICE value in an array over a period of time.
LowestBar	Finds the number of bars ago when the lowest PRICE occurred.
LowestFC	Finds the lowest PRICE value over a period of time using fast method.
LowM	Low price of *N* months ago from the current bar.
LowW	Low price of *N* weeks ago from the current bar.
LowY	Low price of *N* year ago from the current bar.
LRO	The number of bars ago the specified expression was True.
LWAccDis	Calculates the Larry Williams – Accumulation Distribution total.
M	
MACD	Difference between a fast and slow moving average for a specified price.
MassIndex	Warns of an impending direction change.

McClellanOsc	Market breadth based on smoothed difference between the NYSE adv/dec issues.
Median	Gets the median value from a series of values.
MedianArray	Gets the median value from a series of values in an array.
MedianPrice	Calculates the mid-price (median) of the bar.
MFI	Returns the Range divided by Volume.
MidPoint	Calculates the average of the highest and lowest price over a specified period.
MinutesToTime	Converts minutes into HH:MM format.
Mode	The most frequently occurring or repetitive value in a specified period.
ModeArray	The most frequently occurring or repetitive array value in a specified period.
Momentum	Calculates the change in price (overbought/oversold) during a specified period.
MoneyFlow	The positive or negative money flow over a period of bars.
MRO	The number of bars ago that an expression was true.
MyPrice	The average bar price based on the high, low, and close.
N	
Next3rdFriday	Number of days to the next third Friday in the month.
NormCumDensity	Normal Cumulative Density for a specified mean and standard deviation.
NormCumDensityArray	Normal Cumulative Density for a specified mean and standard deviation in an array.
NormDensity	Normal Density (also called distribution) for a specific value.
NormDensityArray	Normal Density (also called distribution) for a specific value in an array.
NormSCDensity	Calculates the standard normal cumulative distribution for a data series.
NthExtremes	Finds the Nth highest and lowest values over a number of bars.
NthExtremesArray	Finds the Nth highest and lowest values of an array.
NthHighest	Finds the Nth highest value of price over a number of bars.
NthHighestArray	Finds the Nth highest value in an array.
NthHighestBar	The bar number of the Nth highest value of price over a number of bars.
NthLowest	Find the Nth lowest value of price over a number of bars.
NthLowestArray	Find the Nth lowest value in an array.
NthLowestBar	The bar number of the Nth lowest value of price over a number of bars.
NumericRank	Calculates the rank of a number in a list.
NumericRankArray	Calculates the rank of a number in an array.
NumUnits	The number of shares to buy based on the amount and minimum lot values used.
O	
OBV	On Balance Volume gauges the buying and selling pressure in the market.
OHLCPeriodsAgo	Calculates the Open, High, Low and Close for the specified periods in the past.
OpenD	Opening price of *N* days ago from the current bar.
OpenM	Opening price of *N* months ago from the current bar.
OpenW	Opening price of *N* weeks ago from the current bar.
OpenY	Opening price of *N* year ago from the current bar.

P	
Parabolic	Parabolic SAR for the current bar.
ParabolicCustom	Parabolic SAR for the current bar using a custom price.
ParabolicSAR	Parabolic SAR for the current bar.
PercentChange	The percent change in price of the current bar over the price length bars ago.
Percentile	Calculates the Percentile (k-th value) of a specified period.
PercentileArray	Calculates the Percentile (k-th value) of a specified period in an array.
PercentR	Identifies occurrences of prices outside this normal trading range.
PercentRank	The rank of a value in a data set as a percentage of the data set.
PercentRankArray	The rank of a value in a data set as a percentage of the array.
Permutation	Number of permutations for N objects from a range of objects.
Pivot	Calculates the value and the number of bars ago a pivot occurred.
PivotHighVS	The specified value of the High Pivot bar with variable strength sides.
PivotHighVSBar	Bars ago that the Pivot High bar, with variable strength sides, occurred.
PivotLowVS	The specified value of the Low Pivot bar with variable strength sides.
PivotLowVSBar	Bars ago that the Pivot Low bar, with variable strength sides, occurred.
PositionProfitCustom	Customized position profit value for current or maximum position profit.
PriceOscillator	Calculates the price oscillator for the current bar.
PriceVolTrend	Calculates the trend based on trade volume.
ProbAbove	The probability that a price will rise or remain above a price target.
ProbBelow	The probability that a price will fall or remain below a price target.
ProbBetween	The probability that a price will be within the specified low and high range.
Q	
Quartile	Calculates the quartile value of a data set for a specified quarter.
QuartileArray	Calculates the quartile value of an array for a specified quarter.
R	
Range	Calculates a bars range buy subtracting the low from the high.
RangeLeader	True if the current bar is considered a range leader.
RateOfChange	Rate of Change determines the magnitude of oscillations based on volatility.
RecentOcc	Returns the number of bars ago a condition was true.
Round2Fraction	The nearest fractional value for a decimal variable.
RSI	Relative Strength Index indicates momentum ranging from 0 to 100.
RSquare	The square of the Pearson product moment correlation coefficient, R.
RSquareArray	The square of the Pearson product moment correlation coefficient, R, based on an array.
S	
ShowLongStop	Adds text to a chart displaying the stop level for a long-side stop.
ShowShortStop	Adds text to a chart displaying the stop level for a short-side stop.
Skew	Calculates the Skewness of a distribution for a set of values.

SkewArray	Calculates the Skewness of a distribution for an array.
SkewOpt	The optimizable skew for a set of values.
SlowD	Slow (smoothed) value used in Stochastic calculations.
SlowDCustom	Slow (smoothed) value used in Stochastic calculations with custom prices.
SlowDCustomOrig	Same as SlowDCustom, using original smoothing methods of Stochastics inventor.
SlowK	Slow (smoothed) value used in Stochastic calculations.
SlowKCustom	Slow (smoothed) value based on custom prices used in Stochastic calculations.
SlowKCustomOrig	Same as SlowKCustom, using original smoothing methods of Stochastics inventor.
SmoothedAverage	Used like Average but provides smoothed value.
Sort2DArray	Sorts an array of two dimensions.
SortArray	Sorts an array of one dimension.
StandardDev	Calculates a standard deviation of values.
StandardDevAnnual	Calculates a standard deviation of values and presents an annualized number.
StandardDevArray	Calculates a standard deviation of values based on an array.
Standardize	Normalized value from a distribution.
StandardizeArray	Calculates a normalized value from an array.
StdDev	The amount prices vary from the mean average (using the same parameters).
StdDevS	Calculates the Sample Standard Deviation value of the specified bar.
StdError	Calculates standard variation around a regression line.
StdErrorArray	Calculates standard variation around a regression line, based on an array.
Stochastic	Calculates all of the stochastic values.
StrColorToNum	The color number of a given color's name.
Summation	Provides the sum total of a series of numbers.
SummationArray	Provides the sum total of an array of values.
SummationFC	Provides the sum total of a series of numbers using the fast method.
SummationIf	Provides the sum total of a series of numbers when a condition is true.
SummationRecArray	Calculates a summation of the reciporical value of array elements.
SummationSqrArray	Calculates a summation of the square of array elements.
SwingHigh	Finds the Swing High price over a series of bars. Returns –1 if none found.
SwingHighBar	Number of the bar ago a Swing High occurred.
SwingIndex	Positive or negative value (+100 to –100) indicating trend direction.
SwingLow	Finds the Swing Low price over a series of bars. Returns –1 if none found.
SwingLowBar	Number of the bar ago a Swing Low occurred.
T	
TimeSeriesForecast	Plots a line through prices to minimize the distance between a line and point.
TimeToMinutes	Converts time in 24-hour format to minutes after midnight.
TL_Exist	True if the specified trendline exists.
TLAngle	Angle of an imaginary trend line between two points on your chart.
TLAngleEasy	Angle of an imaginary trend line between two bars on your chart.
TLSlope	Slope of an imaginary trend line between two points on your chart.

TLSlopeEasy	Slope of an imaginary trend line between two bars on your chart.
TLValue	Price of a target bar based on an imaginary trend line into the future.
TLValueEasy	Price of a target bar based on an imaginary trend line into the future.
TriAverage	Triangular Moving Average weighted on the middle portion of the length.
TrimMean	Mean of the interior portion of a set of data.
TrimMeanArray	Calculates the interior mean value of an array.
TRIX	Triple Smooth Exponential Average
TrueHigh	Returns the high of the current bar, or the close of the previous bar if higher.
TrueLow	Returns the low of the current bar, or the close of the previous bar if lower.
TrueRange	Difference between the TrueHigh and TrueLow values.
TrueRangeCustom	The difference between user specified values based on high, low, and close.
TypicalPrice	Similar to AvgPrice except that it uses an average of the high, low and close.
U	
UlcerIndex	A measure of the stress level related to market condition.
UltimateOscillator	Oscillator based on three different time frames.
V	
VarianceArray	Calculates the estimated variance based on an array.
VariancePS	Calculates the estimated variance.
Volatility	Average of the TrueRange over a specific number of bars.
VolatilityStdDev	The statistical (historical) volatility based on a standard deviation of closes.
VolumeOsc	Difference between a slow and fast period moving average in terms of points.
VolumeROC	Positive or negative value of the likelihood of a continuation in current move.
W	
WAverage	Weighted moving average of the price over a specified number of bars.
WeightedClose	Similar to the AvgPrice function except it gives weight to additional avg close.
X	
XAverage	Weighted moving average of the prices of the last length bars.
XAverageOrig	Weighted moving average of prices using a distinct smoothing method.
Y	
Z	
ZProb	Two-tailed P-value of a Z-test.

APPENDIX C

Reserved Words

ActivityBar Study

AB_AddCell	Adds a cell to an ActivityBar row.
AB_GetCellChar	Returns the character stored in a cell.
AB_GetCellColor	Returns the color of the character stored in a cell.
AB_GetCellDate	Returns the corresponding date of a cell.
AB_GetCellTime	Returns the corresponding time of a cell.
AB_GetCellValue	Returns the extra-value stored in a cell.
AB_GetNumCells	Returns how many cells exist in a row-side.
AB_GetZoneHigh	Returns the value of the top (high) of the ActivityBar zone.
AB_GetZoneLow	Returns the value of the bottom (low) of the ActivityBar zone.
AB_High	Returns the high of the current ActivityBar.
AB_Low	Returns the low of the current ActivityBar.
AB_RemoveCell	Removes a cell from an ActivityBar row.
AB_RowHeight	Returns the cell height from an ActivityBar.
AB_SetActiveCell	Sets a cell-row as the active cell.
AB_SetRowHeight	Changes the current ActivityBar's row-increment value.
AB_SetZone	Set a zone range-box for an ActivityBar side.
ActivityData	References any bar data element (Open, upticks, etc.) of an ActivityBar.
LeftSide	Used with ActivityBars to refer to a cell or zone on the left side of a bar.
RightSide	Used with ActivityBars to refer to a cell or zone on the right side of a bar.

Alerts and Commentary

Alert	Triggers an alert for an indicator, ShowMe, PaintBar, or ActivityBar.
AlertEnabled	True/false expression returning true if the Enable Alert check box is selected.
AtCommentaryBar	True/false expression returning true when Analysis Commentary is applied to a bar.
Cancel	Used in conjunction with Alert to cancel a previously enabled alert.
CheckAlert	Returns true for the last bar when Enable Alert check box is selected.
Commentary	Sends EasyLanguage expression(s) to the Analysis Commentary window.
CommentaryCl	Sends EasyLanguage expression(s) to Analysis Commentary with a carriage return.
CommentaryEnabled	True/false expression returns true when the Analysis Commentary window is open.

Backward Compatibility

Based	Skip word.
BreakEvenStopFloor	Break-even stop floor amount.
CheckCommentary	True/false expression returning true when Analysis Commentary is applied to a bar.
Default	Used in plot statements to set one of its styles to its default value.
DefineCustField	Reserved for future use.
GetSystemName	The name of the trading strategy applied to the chart.
IncludeSignal	Allows the inclusion of a strategy within another strategy.
IncludeSystem	Allows the inclusion of a strategy within another strategy.
Moc	Reserved for future use.
MoneyMgtStopAmt	Money management stop dollar amount
Not	Reserved for future use.
Place	Skip word.
Pob	A synonym for a limit order.
ProfitTargetStop	Profit target stop amount.
Repeat	Reserved for future use.
Screen	Reserved for future use.
Skip	Reserved for future use.
Text	Reserved for backward compatibility with previous EasyLanguage versions.
Today	References the most current bar, even when analyzing intraday bars.
Tomorrow	References the next bar, even when analyzing intraday bars.
Tool_Black	References the color black.
Tool_Blue	References the color blue.
Tool_Cyan	References the color cyan.
Tool_DarkBlue	References the color dark blue.
Tool_DarkBrown	References the color dark brown.
Tool_DarkCyan	References the color dark cyan.
Tool_DarkGray	References the color dark gray.
Tool_DarkGreen	References the color dark green.
Tool_DarkMagenta	References the color dark magenta.
Tool_DarkRed	References the color dark red.
Tool_DarkYellow	References the color dark yellow.
Tool_Green	References the color green.
Tool_LightGray	References the color light gray.
Tool_Magenta	References the color magenta.
Tool_Red	References the color red.
Tool_White	References the color white.
Tool_Yellow	References the color yellow.
TrailingStopAmt	Risk trailing stop dollar amount.
TrailingStopFloor	Risk trailing stop floor amount.
TrailingStopPct	Risk trailing stop percent amount.

Units	Number of assets, options, or futures comprising a specific position leg.
Until	Reserved for future use.
Yesterday	References the previous bar, even when analyzing intraday bars.

Colors

Black	Specifies color Black (numeric value = 1) for plots and backgrounds.
Blue	Specifies color Blue (numeric value = 2) for plots and backgrounds.
Cyan	Specifies color Cyan (numeric value = 3) for plots and backgrounds.
DarkBlue	Specifies color Dark Blue (numeric value = 9) for plots and backgrounds.
DarkBrown	Specifies color Dark Brown (numeric value = 14) for plots and backgrounds.
DarkCyan	Specifies color Dark Cyan (numeric value = 10) for plots and backgrounds.
DarkGray	Specifies color Dark Gray (numeric value = 15) for plots and backgrounds.
DarkGreen	Specifies color Dark Green (numeric value = 11) for plots and backgrounds.
DarkMagenta	Specifies color Dark Magenta (numeric value = 12) for plots and backgrounds.
DarkRed	Specifies color Dark Red (numeric value = 13) for plots and backgrounds.
Green	Specifies color Cyan (numeric value = 4) for plots and backgrounds.
LightGray	Specifies color Light Gray (numeric value = 16) for plots and backgrounds.
Magenta	Specifies color Magenta (numeric value = 5) for plots and backgrounds.
Red	Specifies color Red (numeric value = 6) for plots and backgrounds.
White	Specifies color White (numeric value = 8) for plots and backgrounds.
Yellow	Specifies color Yellow (numeric value = 7) for plots and backgrounds.

Comparison and Loops

Above	Detects when a value crosses over, or becomes greater than another value.
And	Links two true/false expressions together. True if both expressions are true.
Begin	Used to begin a block statement (e.g., If-Then-Else, For loops, While loops).
Below	Detects when a value crosses below, or becomes less than another value.
Cross	Used to detect when values have crossed over/under another value.
Crosses	Used to detect when values have crossed over/under another value.
DownTo	Instructs a loop's counter to decrement and exit the loop at a specified value.
Else	Included in If-Then statements to execute an alternate set of statements.
End	Completes a block of instructions that follow a Begin statement.
False	Assigns a false value to a variable. Checks the status of an expression.
For	Defines a group of instructions executed a predefined number of times.
If	Specifies a condition that must be met to execute a set of instructions.
Or	Links 2 true/false expressions together. True if either expression is true.
Over	Detects when a value crosses above, or becomes greater than another value.
Then	Specifies the action to be executed if an If-Then statement is true.
To	Instructs a For-Loop statement to increment its count by one each iteration.

True	Assigns a true value to a variable. Checks the status of an expression.
Under	Detects when a value crosses under, or becomes less than another value.
While	Defines instructions executed until a true/false expression returns false.

Compiler Directives

#BEGINALERT	A compiler directive including all instructions between #BeginAlert and #End.
#BEGINCMTRY	A compiler directive including all instructions between #BeginCmtry and #End.
#BEGINCMTRYORALERT	A compiler directive including instructions between #BeginCmtryOrAlert and #End.
#END	A compiler directive used to terminate an alert or commentary statement.

Data Information / Fundamental

HistFundExists	Informs if historical fundamental data available for symbol.
SnapFundExists	Informs if snapshot fundamental data available for symbol.

Data Information/General

Ago	References a specified number of bars back already analyzed by EasyLanguage.
Bar	References a specific bar.
BarInterval	Bar interval of data stream currently being analyzed.
Bars	References a specific bar.
BarStatus	Returns 0 for the first tick, 1 for a normal-tick, and 2 for bar-close.
BigPointValue	Dollar amount of 1 full point move.
BoxSize	Box size of Point & Figure chart.
C	Returns the closing price of the bar referenced. (Abbreviation for Close).
Close	Returns the closing price of the bar referenced.
CommodityNumber	Unique number representing particular symbol (optional).
Contract	Specifies the number of units to trade within a trading strategy.
ContractMonth	Refers to the delivery/expiration month of any option, future, or position leg.
Contracts	Specifies the number of contracts/share for a particular order.
ContractYear	Refers to the delivery/expiration year of any option, future, or position leg.
Current	Reserved for future use.
CurrentBar	Bar number of current bar.
D	Returns the closing date of the bar referenced. (Abbreviation for Date).
DailyLimit	Number of stocks/contracts allowed traded in 1 day.
Data	Used to reference information from a specified data stream.
DataCompression	0 for tick, 1 for intra-day, 2 for daily, 3 for weekly, 4 for monthly, 5 for P&F.
DataInUnion	Reserved for future use.
Date	Returns the closing date of the bar referenced.

Day	References a specific bar occurring N days ago.
Days	References a specific bar occurring N days ago.
DeliveryMonth	Delivery month of futures contract.
DeliveryYear	Delivery year of futures contract.
DownTicks	Reserved for backward compatibility. Replaced with DnVolume.
ExpirationDate	Returns the expiration/delivery date of an option, future, or position leg.
FirstNoticeDate	Returns the first notice date of a futures contract.
GetExchangeName	The name of the exchange for a symbol.
GetSymbolName	Name of the symbol study currently analyzing.
H	Returns the highest price of the bar referenced. (Abbreviation for High)
High	Returns the highest price of the bar referenced.
I	Number of contracts outstanding at the close of a bar. (Abbr. for OpenInt)
L	Returns the lowest price of the bar referenced. (Abbreviation for Low).
LastTradingDate	Refers to the last day an option, future, position leg, or asset may be traded.
Low	Returns the lowest price of the bar referenced.
Market	Order type referring to the opening price of the next bar.
MaxBarsBack	Maximum number of reference bars (buffer) needed before study can plot.
MaxBarsForward	Number bars allocated (by charting) to the right of the chart.
MinMove	Minimum tick movement of a symbol.
Next	Used with Bar to reference the next bar in a trading strategy.
O	Returns the opening price of the bar referenced. (Abbreviation for Open)
Open	Returns the opening price of the bar referenced.
OpenInt	Returns the number of contracts outstanding at the close of the bar referenced.
Point	Returns the minimal interval value a symbol can move.
Points	Returns the minimal interval value a symbol can move.
PointValue	Dollar amount of 1 point move.
PriceScale	Price scale of stock/future symbol (inverted for EasyLanguage).
RevSize	Reversal size of Point & Figure chart
Sess1EndTime	Ending time of first session.
Sess1FirstBarTime	Time when first bar in morning session completed.
Sess1StartTime	Starting time of first session
Sess2EndTime	Ending time of second session.
Sess2FirstBarTime	Time when first bar in second session completed.
Sess2StartTime	Starting time of second session.
StartDate	Reserved for future use.
T	Returns the closing time of the bar referenced. (Abbreviation for Time),
This	Used with Bar to reference the current bar.
Ticks	Reserved for backward compatibility. Replaced with Volume.
Time	Closing time of the bar in charting or specified time interval in a grid.
UnionSess1EndTime	Latest session 1 end time of all data in a multi-data chart.
UnionSess1FirstBar	Earliest session 1 first bar time of all data in a multi-data chart.

UnionSess1StartTime	Earliest session 1 start time of all data in a multi-data chart.
UnionSess2EndTime	Latest session 2 end time of all data in a multi-data chart.
UnionSess2FirstBar	Earliest session 2 first bar time of all data in a multi-data chart.
UnionSess2StartTime	Earliest session 2 start time of all data in a multi-data chart.
UpTicks	Reserved for backward compatibility. Replaced with UpVolume.
V	Number of shares/contracts traded for the bar referenced. (abbr. for Volume).
Volume	Returns the number of shares or contracts traded for the bar referenced.

Date and Time

CurrentDate	Computer or datafeed current calendar date.
CurrentTime	Computer or datafeed current time, in 24 hr format.
DateToJulian	Converts calendar date to Julian date.
DayOfMonth	Day's date on specified calendar date.
DayOfWeek	Day of week (0 for Sun., 1 for Mon., ..., 6 for Sat.) on specified calendar date.
EL_DateStr	Returns a string composed of the month,day,year passed.
Friday	Specifies day of the week Friday (numeric value = 5).
JulianToDate	Converts Julian date to calendar date.
LastCalcJDate	Julian date of last completed bar.
LastCalcMMTime	Time of last completed bar, in minutes since midnight.
Monday	Specifies day of the week Monday (numeric value = 1).
Month	Month on specified calendar date, from 1 to 12
Saturday	Specifies day of the week Saturday (numeric value = 6).
Sunday	Specifies day of the week Sunday (numeric value = 0).
T	Returns the closing time of the bar referenced. (Abbreviation for Time)
Thursday	Specifies day of the week Thursday (numeric value = 4).
Time	Closing time of the bar referenced.
Tuesday	Specifies day of the week Tuesday (numeric value = 2).
Wednesday	Specifies day of the week Wednesday (numeric value = 3).
Year	Year on specified calendar date, in short form (last 2 digits of year)

Declaration

Array	Used to declare an array.
Arrays	Used to declare an array.
Input	Declares custom words to behave as constants throughout an analysis technique.
Inputs	Declares custom words to behave as constants throughout an analysis technique.
Numeric	Defines an input as a numeric expression.
NumericArray	Defines an input as a numeric array.

NumericArrayRef	Defines an input as a numeric function-modifiable array.
NumericRef	Allows the code to pass a Numeric variable so it can be modified by the function.
NumericSeries	Defines an input as a numeric series expression.
NumericSimple	Defines an input as a numeric simple expression.
String	Defines an input as a string expression.
StringArray	Defines an input as a string array.
StringArrayRef	Defines an input as a string function-modifiable array.
StringRef	Allows the code to pass a Text-String variable so it can be modified by the function.
StringSeries	Defines a function's input as a string series expression.
StringSimple	Defines a function's input as a string simple expression.
TrueFalse	Defines an input as a true/false expression.
TrueFalseArray	Defines an input as a true/false array.
TrueFalseArrayRef	Defines an input as a true/false function-modifiable array.
TrueFalseRef	Allows the code to pass a TrueFalse variable so it can be modified by the function.
TrueFalseSeries	Defines an input as a true/false series expression.
TrueFalseSimple	Defines an input as a true/false simple expression.
Var	Declares words to recognize as variables throughout your analysis technique.
Variable	Declares words to recognize as variables throughout your analysis technique.
Variables	Declares words to recognize as variables throughout your analysis technique.
Vars	Declares words to recognize as variables throughout your analysis technique.

DLL

ARRAYSIZE	Reserved for use with ELKIT32.DLL.
ARRAYSTARTADDR	Reserved for use with ELKIT32.DLL.
BOOL	Reserved for use with ELKIT32.DLL.
BYTE	Reserved for use with ELKIT32.DLL.
CHAR	Reserved for use with ELKIT32.DLL.
DEFINEDLLFUNC	Reserved for use with ELKIT32.DLL to declare a DLL.
DOUBLE	Reserved for use with ELKIT32.DLL.
DWORD	Reserved for use with ELKIT32.DLL.
FLOAT	Reserved for use with ELKIT32.DLL.
INT	Reserved for use with ELKIT32.DLL.
LONG	Reserved for use with ELKIT32.DLL.
LPBOOL	Reserved for use with ELKIT32.DLL.
LPBYTE	Reserved for use with ELKIT32.DLL.
LPDOUBLE	Reserved for use with ELKIT32.DLL.
LPDWORD	Reserved for use with ELKIT32.DLL.
LPFLOAT	Reserved for use with ELKIT32.DLL.
LPINT	Reserved for use with ELKIT32.DLL.
LPLONG	Reserved for use with ELKIT32.DLL.

LPSTR	Reserved for use with ELKIT32.DLL.
LPWORD	Reserved for use with ELKIT32.DLL.
MULTIPLE	Reserved for use with ELKIT32.DLL.
POINTER	Reserved for use with ELKIT32.DLL.
UNSIGNED	Reserved for use with ELKIT32.DLL.
VARSIZE	Reserved for use with ELKIT32.DLL.
VARSTARTADDR	Reserved for use with ELKIT32.DLL.
VOID	Reserved for use with ELKIT32.DLL.
WORD	Reserved for use with ELKIT32.DLL.

Math and Trig

AbsValue	Absolute value of num.
Arctangent	Arctangent value of num, in degrees.
AvgList	Average of nums in list.
Ceiling	Lowest integer greater than num.
Cosine	Cosine value of num, in degrees.
Cotangent	Cotangent value of num, in degrees.
ExpValue	Exponential value of num.
Floor	Highest integer less than num.
FracPortion	Fractional portion of num.
IntPortion	Integer portion of num.
Log	Natural logarithm of num.
MaxList	Highest value num in list.
MaxList2	Second highest value num in list.
MinList	Lowest value num in list.
MinList2	Second lowest value num in list.
Mod	Remainder of num/divisor.
Neg	Absolute negative of num.
NthMaxList	Nth highest value num in list.
NthMinList	Nth lowest value num in list.
Pos	Absolute positive of num.
Power	Num raised to the Nth power.
Random	Returns a pseudo-random number.
Round	Num rounded to nearest precision.
Sign	1 for positive num, -1 for negative num and 0 for 0.
Sine	Sine value of num, in degrees.
Square	Square of num.
SquareRoot	Square root of num.

SumList	Sum of all nums in list.
Tangent	Tangent of num degrees.

Messaging

Pager_DefaultName	Default subscriber name.
Pager_Send	Sends text message str_Msg to str_Name (if pager module enabled).

Multimedia

AddToMovieChain	Appends movie file to end of movie chain.
GetCDRomDrive	Drive letter of first CD-ROM found.
MakeNewMovieRef	Creates new movie reference number.
PlayMovieChain	Queues then plays movies in movie chain.
PlaySound	Plays sound from file.

Output

ClearDebug	Clears the contents of the Print Log tab.
File	Sends information to a specified file from a print statement.
FileAppend	Appends text string to file.
FileDelete	Deletes the specified file.
MessageLog	Sends EasyLanguage expression(s) to the Print Log tab.
Print	Sends information to the Print Log, a specified file, or a printer.
Printer	Sends information to a printer from a print statement.

Plotting

GetBackgroundColor	Current chart background color (see documentation for color values).
GetPlotBGColor	Returns the background color of a cell for an analysis technique.
GetPlotColor	Returns the numeric color value of a chart's plot line or grid's foreground.
GetPlotWidth	Returns the width value of a plot line in a chart.
NoPlot	Removes a plot from the current bar in a chart or cell in a grid.
Plot	References the value of a plot.
Plot1	References the value of a plot.
Plot2	References the value of a plot.
Plot3	References the value of a plot.
Plot4	References the value of a plot.
PlotPaintBar	Plots a range of values inside the current bar in a chart.
PlotPB	Plots a range of values inside the current bar in a chart.

SetPlotBGColor	Assigns a specified background color to a grid containing an indicator.
SetPlotColor	Assigns the color value (color) to the plot specified by (num).
SetPlotWidth	Modifies the thickness of an indicator's plot line.

ProbabilityMaps

PM_GetCellValue	Returns the intensity value of a cell at the specified column and price location.
PM_GetNumColumns	Returns the number of columns in a probability map array.
PM_GetRowHeight	Returns the height or increment of the rows in a ProbabilityMap study.
PM_High	Returns the value of the upper range of a ProbabilityMap grid.
PM_Low	Returns the value of the lower range of a ProbabilityMap grid.
PM_SetCellValue	Sets the location and intensity of ProbabilityMap cells.
PM_SetHigh	Sets the upper range value of a ProbabilityMap.
PM_SetLow	Sets the lower range value of a ProbabilityMap.
PM_SetNumColumns	Sets the number of columns in a probability map array.
PM_SetRowHeight	Sets the height of the rows for a ProbabilityMap grid.

Product Information

BlockNumber	Unique Security Block number.
CurrentDate	Computer or data current calendar date.
CurrentTime	Computer or data current time, in 24 hr format.
CustomerID	Unique customer ID number.
EasyLanguageVersion	Number representing EasyLanguage implementation version.
Product	Number representing Omega product currently being used.

Skip Words

A	Skip word.
An	Skip word.
At	Skip word.
By	Skip word.
Does	Skip word.
Is	Skip word.
Of	Skip word.
On	Skip word.
Than	Skip word.
The	Skip word.
Was	Skip word.

Strategy Orders

All	Specifies all shares/contracts are to be sold/covered when exiting a position.
At$	Anchors exit prices to the bar where the entry order was placed.
Bar	References a specific bar.
Buy	Initiates a long position. Covers any short positions & reverses your position.
BuyToCover	Used in trading strategies to partially or completely cover short positions.
Entry	Ties an exit to an entry order in a strategy.
From	Skip word.
Higher	Synonym for stop or limit orders depending on the context used within a strategy.
Limit	A limit order meaning 'or higher' or 'or lower', depending on the context.
Lower	Synonym for stop or limit orders depending on the context used within a strategy.
Market	Order type referring to the opening price of the next bar.
Next	Used with Bar to reference the next bar in a trading Strategy.
Point	Returns the minimal interval value a symbol can move.
Points	Returns the minimal interval value a symbol can move.
Sell	Used in trading strategies to partially or completely liquidate a long position.
SellShort	Initiates a short position, closes any open long positions & reverses position.
SetBreakeven	Sets up a break-even stop.
SetDollarTrailing	Sets up a dollar-trailing stop.
SetExitOnClose	Sets the Exit on Close stop to true.
SetPercentTrailing	Sets up a percent-trailing stop.
SetProfitTarget	Sets up a profit-target stop.
SetStopContract	Sets the builtin stops to execute on a contract basis.
SetStopLoss	Sets up a stop-loss stop.
SetStopPosition	Sets the built-in stops to execute on a position basis.
SetStopShare	Sets the built-in stops to execute on a share basis.
Share	Used to specify the number of contracts/shares for a particular order.
Shares	Used to specify the number of contracts/shares for a particular order.
Stop	A stop order meaning 'or higher' or 'or lower', depending on the context.
This	Used with Bar to reference the current bar.
Total	Number of shares/contracts to exit from a position created by pyramiding.

Strategy Performance

AvgBarsLosTrade	Average number of bars in closed-out losing trades.
AvgBarsWinTrade	Average number of bars in closed-out winning trades.
AvgEntryPrice	Average price of all currently open entries.
CurrentContracts	Number of contracts currently open.
CurrentEntries	Number of entries currently open.
CurrentShares	Number of shares currently open.

GrossLoss	Cumulative dollar total of all closed-out losing trades.
GrossProfit	Cumulative dollar total of all closed-out winning trades.
I_AvgEntryPrice	Average of applied strategy's open entries.
I_ClosedEquity	Applied strategy's total net profit.
I_CurrentContracts	Number of contracts applied strategy has currently bought/sold.
I_MarketPosition	Applied strategy's current market position: 1 = long, -1 = short, 0 = flat.
I_OpenEquity	Applied strategy's total net profit + open position Profit/Loss.
LargestLosTrade	Dollar amount of largest closed-out losing trade.
LargestWinTrade	Dollar amount of largest closed-out winning trade.
MaxConsecLosers	Longest chain of consecutive closed-out losing trades.
MaxConsecWinners	Longest chain of consecutive closed-out winning trades.
MaxContractsHeld	Maximum number of contracts held at any one time.
MaxIDDrawDown	True dollar amount needed to sustain largest equity dip.
NetProfit	Cumulative dollar total of all closed-out trades.
NumLosTrades	Total count of closed-out losing trades.
NumWinTrades	Total count of closed-out winning trades.
PercentProfit	Percentage of all closed-out winning trades.
TotalBarsLosTrades	Total number of bars in closed-out losing trades.
TotalBarsWinTrades	Total number of bars in closed-out winning trades.
TotalTrades	Number of all closed-out trades in the life of a strategy.

Strategy Position

BarsSinceEntry	Bars since initial entry of position, num position(s) ago.
BarsSinceExit	Bars since position closed-out, num position(s) ago.
CurrentContracts	Number of contracts currently open.
CurrentEntries	Number of entries currently open.
CurrentShares	Number of shares currently open.
EntryDate	Date of entry, num position(s) ago.
EntryPrice	Price of entry, num position(s) ago.
EntryTime	Time of entry of position, num position(s) ago.
ExitDate	Date when position closed-out, num position(s) ago.
ExitPrice	Exit price of closed-out entry, num position(s) ago.
ExitTime	Time when last entry closed-out, num position(s) ago.
MarketPosition	Market position (1 = long, -1 = short, 0 = flat) of num position(s) ago.
MaxContracts	Max contracts held during num position(s) ago.
MaxEntries	Max entries open during life of position, num position(s) ago.
MaxPositionLoss	Dollar amount of largest loss during position, num position(s) ago.
MaxPositionProfit	Dollar amount of largest gain during position, num position(s) ago.
OpenPositionProfit	Profit/Loss of current open position.
PositionProfit	Profit/Loss of position, num position(s) ago.

Strategy Properties

Commission	Commission per stock/contract/transaction.
GetStrategyName	The name of the trading strategy which applied to the chart.
Margin	Margin of futures contract.
Slippage	Slippage per contract.

Text Drawing

GetBackGroundColor	Current chart background color (see documentation for color values).
Text_Delete	Deletes the specified text object.
Text_GetColor	Returns the color of the specified text object (see documentation for color values).
Text_GetDate	Returns the date axis value of the specified text object.
Text_GetFirst	First created text object of the specified type (see documentation for pref types).
Text_GetHStyle	Returns the horizontal style of the text object (see documentation for style values).
Text_GetNext	Text object created after the specified object (see documentation for pref types).
Text_GetString	Text stored in the specified text object.
Text_GetTime	Time axis value of text object.
Text_GetValue	Price axis value of text object.
Text_GetVStyle	Returns the vertical style of the text object (see documentation for style values).
Text_New	Draws text object at a specified value on a specified date and time.
Text_SetColor	Changes the color of the specified text object (see documentation for color values).
Text_SetLocation	Moves the specified text object.
Text_SetString	Changes the text string of the specified text object.
Text_SetStyle	Sets horiz/vert position of the text object (see documentation for horiz/vert values).

Text Manipulation

InStr	Location of string2 within string1.
LeftStr	Leftmost portion of string.
LowerStr	Lowercase copy of string.
MidStr	Arbitrary slice of the specified string, starting at a position for *n* characters.
NewLine	Carriage return/linefeed useful for commentary/file output strings.
NumToStr	Converts a numeric to a string with decimal places.
RightStr	Rightmost portion of string.
Spaces	Creates a string of empty spaces, used for padding output.
StrLen	Number of characters in the specified string.
StrToNum	Numerical value of a string, zero if the string is not numeric.
UpperStr	Uppercase copy of the specified string.

Trendline Drawing

GetBackGroundColor	Current chart background color (see documentation for color values).
TL_Delete	Deletes a trendline and recycles its reference.
TL_GetAlert	Returns the indicated trendline’s alert type (see documentation for alert values).
TL_GetBeginDate	Date of trendline's start point.
TL_GetBeginTime	Bar time of trendline's start point.
TL_GetBeginVal	Price axis value at trendline's start point.
TL_GetColor	Trendline's color value (see documentation for color values).
TL_GetEndDate	Date of trendline's end point.
TL_GetEndTime	Bar time of trendline's end point.
TL_GetEndVal	Price value at trendline's end point .
TL_GetExtLeft	True if trendline is extended left, False otherwise.
TL_GetExtRight	True if trendline is extended right, False otherwise.
TL_GetFirst	First created trendline of the specified type (see documentation for pref types).
TL_GetNext	Next trendline created (see documentation for pref types).
TL_GetSize	Thickness of trendline (see documentation for size values).
TL_GetStyle	Trendline's style value (see documentation for style values).
TL_GetValue	Price value at a specified date and time along trendline's projection.
TL_New	Creates a new trendline with listed start and end points.
TL_SetAlert	Sets trendline's alert value (see documentation for alert values).
TL_SetBegin	Sets the start point of a trendline to a specific Date and Time.
TL_SetColor	Sets color of a trendline (see documentation for color values).
TL_SetEnd	Sets the end point of trendline to a specific Date and Time.
TL_SetExtLeft	Sets or clears an indefinite leftward extention of trendline.
TL_SetExtRight	Sets or clears an indefinite rightward extention of trendline.
TL_SetSize	Sets thickness/size of trendline (see documentation for size values).
TL_SetStyle	Sets the trendline line style (see documentation for style values).
Tool_Dashed	Assigns a dashed line to a drawing object.
Tool_Dashed2	Assigns a dashed2 line to a drawing object.
Tool_Dashed3	Assigns a dashed3 line to a drawing object.
Tool_Dotted	Assigns a dotted line to a drawing object.
Tool_Solid	Assigns a solid line to a drawing object.

INDEX